TERADYNE

Programmer's Manual
V2.7 Version

560-539-08
Revision A

© **1991, 1992, 1993, 1994, 1995 Teradyne, Inc.**

This documentation contains trade secrets and confidential information furnished pursuant to a license to the user from Teradyne, Inc. Use and reproduction of these documents are restricted under the terms of the license.

Teradyne, Inc.
30801 Agoura Road
Agoura Hills, California 91301
U.S.A.

Table of Contents

Chapter 1.1: Programming Overview 1.1.1
Chapter 1.2: Creating Pin Database Files 1.2.1
Chapter 1.3: Creating Channel Database Files 1.3.1
Chapter 1.4: Creating MUT Database Files 1.4.1
Chapter 1.5: Creating Specification Database Files 1.5.1
Chapter 1.6: Creating Level Database Files 1.6.1
Chapter 1.7: Creating Waveform Database Files 1.7.1
Chapter 1.8: Creating Pattern Database Files 1.8.1
Chapter 1.9: Creating Calibration Database Files 1.9.1
Chapter 1.10: Creating Workspace Database Files 1.10.1
Chapter 1.11: Creating Instance Database Files 1.11.1
Chapter 1.12: Creating a Flow Control Module 1.12.1
Chapter 1.13: Overlay Level and Waveform Files 1.13.1
Chapter 2.1: Programmer's Guide to the Hardware 2.1.1
Chapter 2.2: Timing Guidelines 2.2.1
Chapter 3.1: Using Templates 3.1.1
Chapter 3.2: Customizing Templates 3.2.1
Chapter 4.1: Memory Fail Processing (MFP) Overview 4.1.1
Chapter 4.2: Memory Fail Processing (MFP) Templates 4.2.1
Chapter 4.3: Memory Fail Processing (MFP) High-Level Programming 4.3.1
Chapter 4.4: Memory Fail Processing (MFP) Low-Level Programming 4.4.1
Chapter 5.1:Using Dual-MPG Mode 5.1.1
Index I.1

Chapter 1.1: Programming Overview

Introduction

This document provides essential programming guidelines for the J990 Series Test systems. It covers the steps required to create a basic test program, including examples.

For the most part, this section is a set of step-by-step instructions for programming and running a J990 test program. In a logical sequence, each chapter in this section covers one of the steps.

Basic Programming Sequence

The recommended process for developing a J990 test program is described below.

1. Create a set of *database* files. This can be done by creating ASCII database files and translating them to their binary equivalents or creating the binary database directly with IG900+ tools.[1]
2. Create the *program flow control* file and compile it to create executable code for linking into the test program.
3. Create a *workspace* file that groups the databases and files to form a test program.
4. Create a *makefile* to help facilitate program changes.
5. Load and link the program modules and databases.
6. Use the IG900+ executive software and IG900+ tools to run the test program, evaluate results, and debug the test program.

---

1. Even though some binary databases can be created through the use of the IG900+ tools, this document advocates the use of ASCII database programming. With ASCII database programming, programs can be created and modified within a TTY terminal or window. This method also creates a *source code* master of the test program.

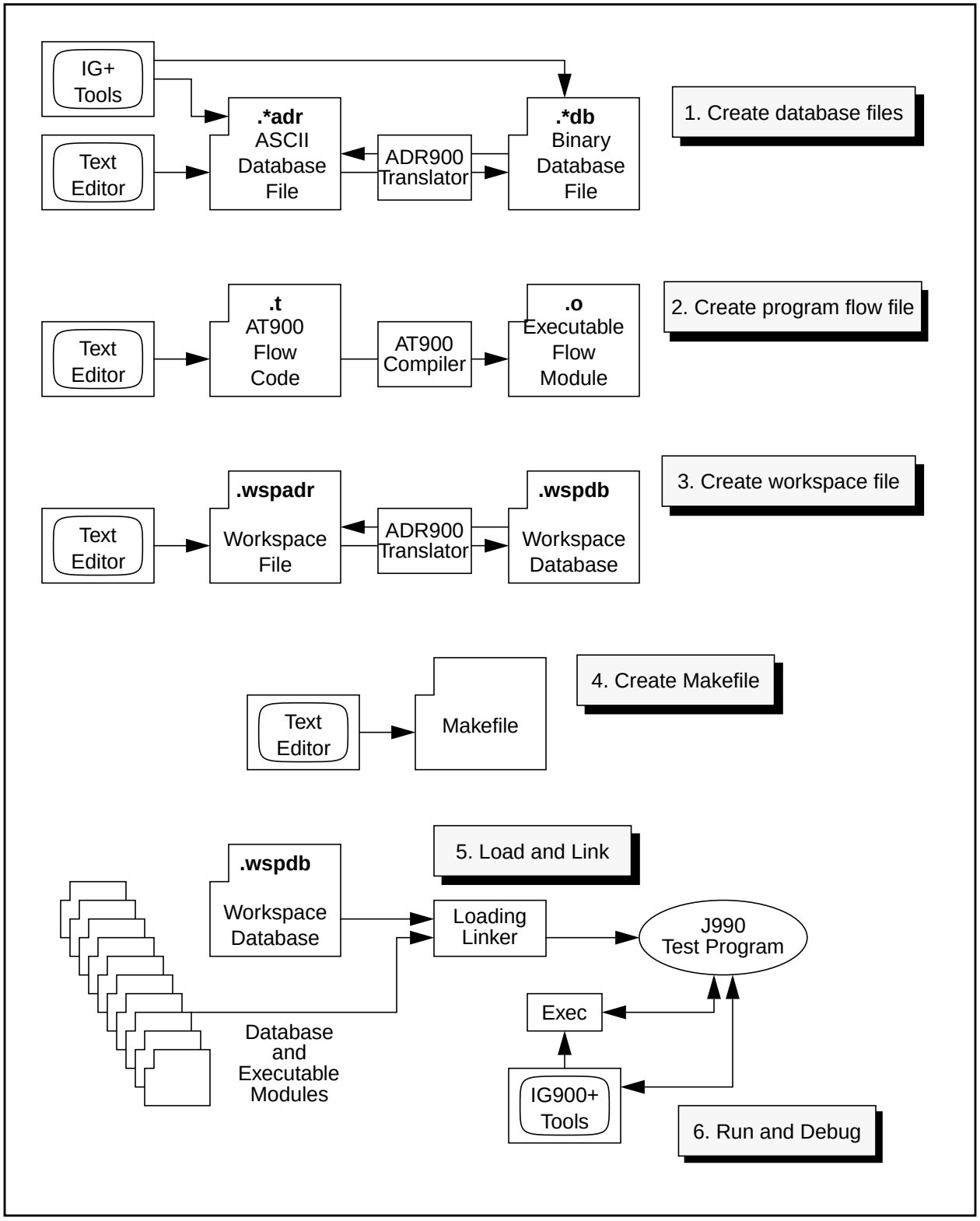


Figure 1.1-1. Illustration of Programming Steps

Programming Process

1. **Create database files**

   Database files are created by editing ASCII files (ASCII Database Representation or ADR files) then translating them with the **adr900** command to create binary data for the test program to use. Alternatively, some of the binary database files can be created directly with IG900+ tools. These databases contain hardware setups and device test conditions for the program to use, including:

   - ❒ Pin and channel mapping
   - ❒ Memory under test description
   - ❒ Device specifications
   - ❒ Timing waveforms and level equations
   - ❒ Test patterns and data generation
   - ❒ Calibration conditions

   ASCII database files use a *.*adr* extension and binary database files use a *.*db* extension. All data sent to the test system hardware, except the test instructions (instances), must be translated to a binary database format.

2. **Create program flow file**

   Flow control is written in the AT900 language, which is a Teradyne superset of C++. It is put into a "T" module (i.e. a file with extension *.t*), then compiled with **at900** command to generate a binary file (*.o*), which will get linked into the program. The flow file determines the test sequence and sorting/binning that takes place.

3. **Create workspace file**

   The workspace file contains a list of the database names and executable modules that make up the test program. It is the handle used by the IG900+ Executive and IG900+ Tools for manipulating the test program. The workspace file is written in ADR syntax and translated with the **adr900** command.

4. **Create a Makefile**

   The makefile specifies the commands to automatically translate and compile the source modules to create new binary files whenever the source code changes. It also contains file dependencies. This facilitates remaking the dependent modules when other modules change. Makefiles are ASCII files created by the programmer that are used by the Unix **make** program.

5. **Load and Link the Test Program**

   Once all the database and executable modules are created, including the workspace and makefile, the program is ready to be linked and loaded. Linking occurs automatically when the program is loaded. During loading, all equations are resolved in the databases, and some tester RAM hardware that can be set before job execution begins.

6. **Run and Debug the Program**

   Use the IG900+ Executive to run the test program, set up test result datalogging, evaluate the results, and view summary data. The IG900+ Tools can be used to help debug the program by modifying databases in the job, re-executing tests, characterizing device specifications, and so on.

Creating Database Files

Several of the files that make up the test program are database files. Typically, database files are created as ADR files, then translated to create binary database files for the test program. Any database file can be created with this method using a standard text editor (vi, textedit, etc.), or using the ADRedit tool. Some types of binary database files can be created directly using IG900+ tools.

Standard Text Editors (vi, textedit tool, etc.):

- ❒ can create all ADR database file types

ADRedit tool:

- ❒ can create all ADR database file types
- ❒ can create all binary database types by direct translation of ADR files

Specsheet tool:

- ❒ can create .specadr and .specdb files (Specification databases)

Calcsheet tool:

- ❒ can create .leveladr, .leveldb, .waveadr, .wavedb files (Level and Waveform Equation databases)

Patedit tool:

- ❒ can create .svmadr and .svmdb files (Pattern Control databases)

Insted tool:

- ❒ can create .insadr files (Instance databases)

Components of a Test Program

Each test program consists of several files with various file extensions. To better understand the process of creating a complete program, the following list describes each of the file types that can occur and describes their contents. Within each database file, there is one or more *sets* which are database objects. For each of these file types, the *adr* version is the ASCII version and the *db* version is the translated binary form.

.pinadr/.pindb

Files with these extensions are pin database files that contain a list of pin types and pin groups. These files are used by other modules which depend on it. Database objects defined within these files are called *pinsets*.

.chanadr/.chandb

Files with these extensions are channel database files that contain the site/device pin mapping to test system channels (a representation of how the DIB is wired). Database objects defined within these files are called *chansets*.

.mutadr/.mutdb

Files with these extensions are memory-under-test database files containing a description of the device. They are ultimately used by the IG900+ Executive to program the address and data path within the test system, including address scrambling and topo inversion. Database objects defined within these files are called *mutsets*.

.insadr

Files with this extension are Instance database files containing the grouping of templates and parameters used by the test program. Instance files are used only in their ASCII form, no db equivalents exist.

main.t/main.o

The main program flow is provided in files with this name. Test instances are executed in the main file and binning and sorting is controlled from these files.

.specadr/.specdb (for DC conditions or levels)

Files with these extensions are specification database files which contain all the level specs and level values for the job. Database objects defined within these files are called *specsets*.

.leveladr/.leveldb

Files with these extensions are level equation database files which contain the level equations for calculating the voltages and currents at the device pins. Database objects defined within these files are called *eqsets*.

.specadr/.specdb (for AC conditions or timing)

Files with these extensions are specification database files which contain all the timing specs and timing values (all dash numbers) for the job. Database objects defined within these files are called *specsets*.

.waveadr/.wavedb

Files with these extensions are waveform equation database files which contain the timing equations used to generate edgesets. Database objects defined within these files are called *eqsets*.

.svmadr/.svmdb

Files with these extensions are patterns database files which contain all the patterns used by the program. These include functional patterns (march, march diagonal, address complement, page scan, etc.) and supplementary patterns (looping pattern, etc.). Database objects defined within these files are called *svmsets*.

.dgenadr/.dgendb

Files with these extensions are data generator database files that specify the datagen hardware sets that will become part of the pattern objects. Database objects defined within these files are called *dgensets*.

.startadr/.startdb

Files with these extensions are pattern start database files containing all the pattern label references that determine where and how to start the patgen. Database objects within these files are called *startsets*.

.srcadr/.srcdb

Files with these extensions are source select ram database files used with pattern objects. Database objects within these files are called *srcsets*.

.caladr/.caldb

Files with this extension are calibration database files which contain the conditions for system calibration. Database objects within these files are called *calsets*.

.wspadr/.wspdb

Files with this extension are workspace database files used to reference all modules and databases and their contents. The job is loaded and linked by using a list of objects within this file.

Makefile

A makefile is a Standard UNIX makefile used to control compilation and translation of source code files and their interdependency.

Database Grouping

Database objects (*sets*) are sometimes grouped to create higher level objects. This distinction between lower level databases and higher level objects is important to understand before actually creating the databases.

- The lower level databases for *specs* and *level equations* are grouped to form a higher level object called a *level*.

  level = specset + eqset

- The lower level databases for *specs* and *waveform equations* are grouped to form a higher level object called a *waveform*.

  waveform = specset + eqset

- The lower level databases for *dgen, svm, mut, src,* and *start* are grouped to form a higher level object called a *pattern*.

  pattern = dgenset + svmset + mutset + srcset + startset

These higher level objects and the remaining databases are all grouped together to create a test program. The combination of databases and test program definitions is performed by the workspace.

This is illustrated by the figure which follows. The lines in the illustration are defined by the workspace.

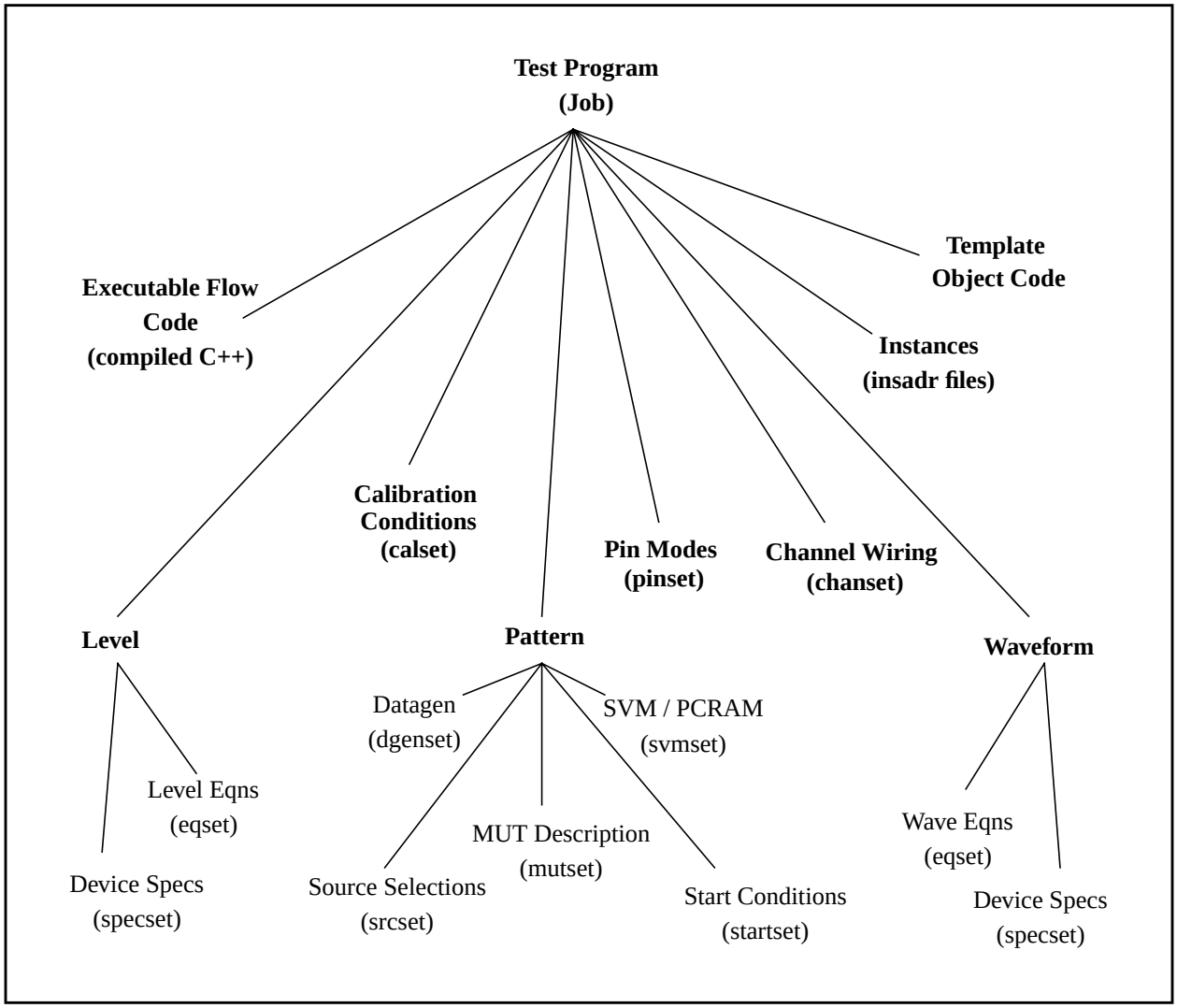


Figure 1.1-2 Test Program Objects and Database Structure

The Executive and Related Software Processes

Test program execution is controlled by the IG900+ Executive. The executive runs the test program, and manages the software interface (datalog, summary, ASCII Commands, etc.). Figure 1.1-3 shows the operation of the IG900+ Executive.

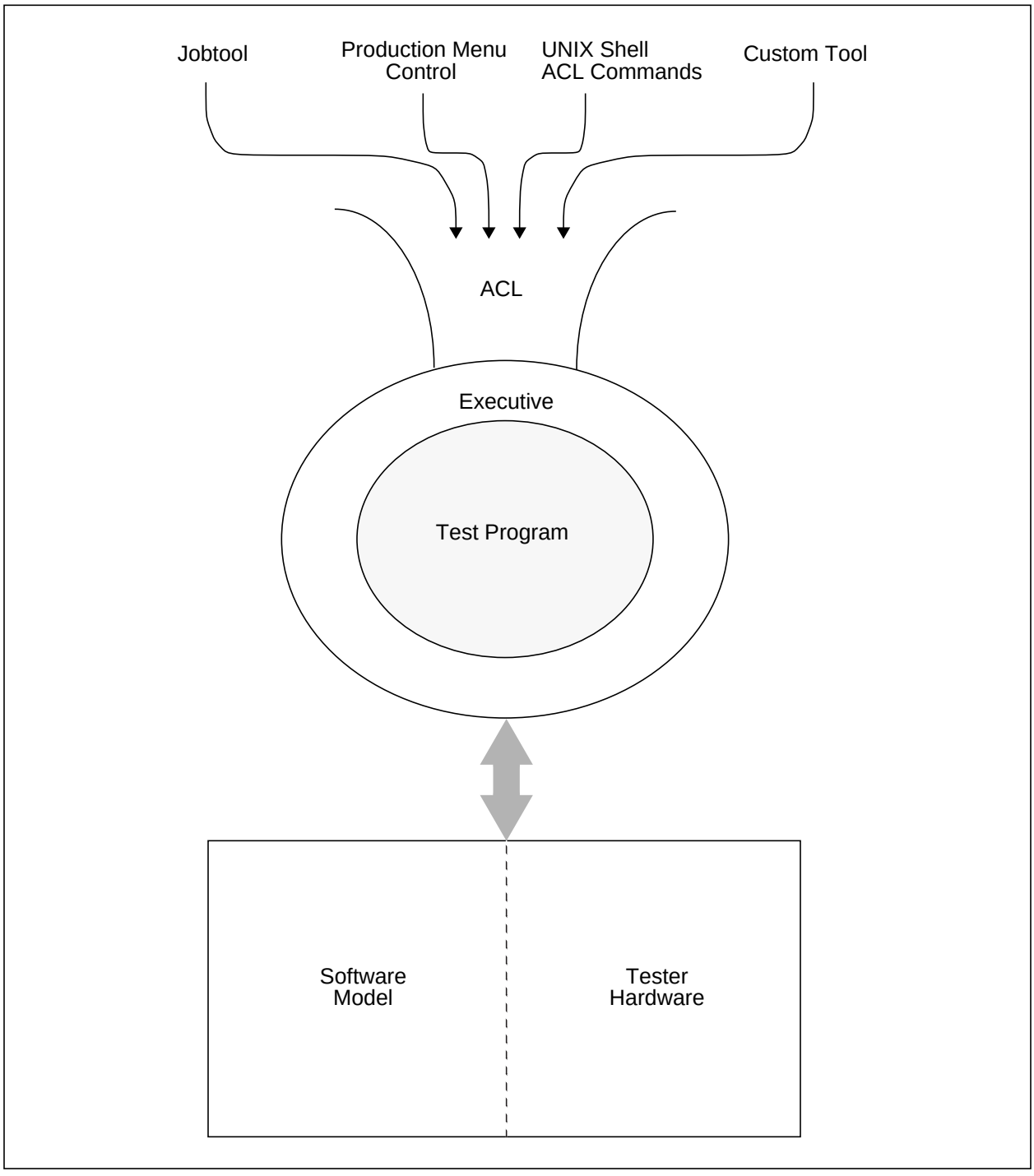


Figure 1.1-3 IG900+ Executive

Program execution actually begins within the executive, before passing to the job_main segment of the test program. At the end of test program, control returns to the executive.

ASCII Control Language (ACL) is used to issue executive commands. Jobtool (or demo_jobtool), the Production Menu Control (PMC) interface, UNIX shell commands (acl commands), and any custom tools, all operate by issuing ACL commands to the executive.

The executive communicates with test system hardware with read and write commands (there may be intermediary layers). These reads and writes may occur directly to the hardware or to the tester software model. The tester model is a software simulation of the tester address space in memory. When operating offline, the executive can operate against the model, allowing some debugging to be completed before test system time is consumed.

Subsequent Chapters

Each chapter of this document describes one step in the process of developing the device test program. For the purposes of this manual, these steps are:

1. Create a pin database
2. Create a channel map database
3. Create a MUT database.
4. Create the specification databases
5. Create the level databases.
6. Create the waveform databases
7. Create the pattern databases
8. Create the calibration conditions database
9. Create instance databases
10. Create the program flow file
11. Create the workspace database
12. Create the program Makefile.
13. Load and run the test program.
14. Debug the test program.

Additional Reading and Reference Material

Teradyne Documentation

The following manuals are additional Teradyne documents that may be of special interest to the programmer.

J990 Series User's Manual
IG900+ Tools Manual
ADR Reference Manual
AT900 Compiler Manual
Templates Reference Manual
Access Reference Manual

Sun Documentation

560-253-01	SunOS V4.1.1 Beginners's Guide

Note: Sun Microsystems offers a full array of documentation and updates for its products. A complete list may be obtained through Sun Microsystems, Inc.

Other Books

C by Dissection, The Essentials of C Programming, Second Edition, Al Kelley & Ira Pohl, The Benjamin/Cummings Company, Inc., 1992.

The Annotated C++ Reference Manual, Margaret A. Ellis & Bjarne Stroustrup AT&T Bell Laboratories, Addison-Wesley Publishing, 1990.

C++ Primer, Stanley B. Lippman, AT&T Bell Laboratories, Addison-Wesley Publishing, 1990.

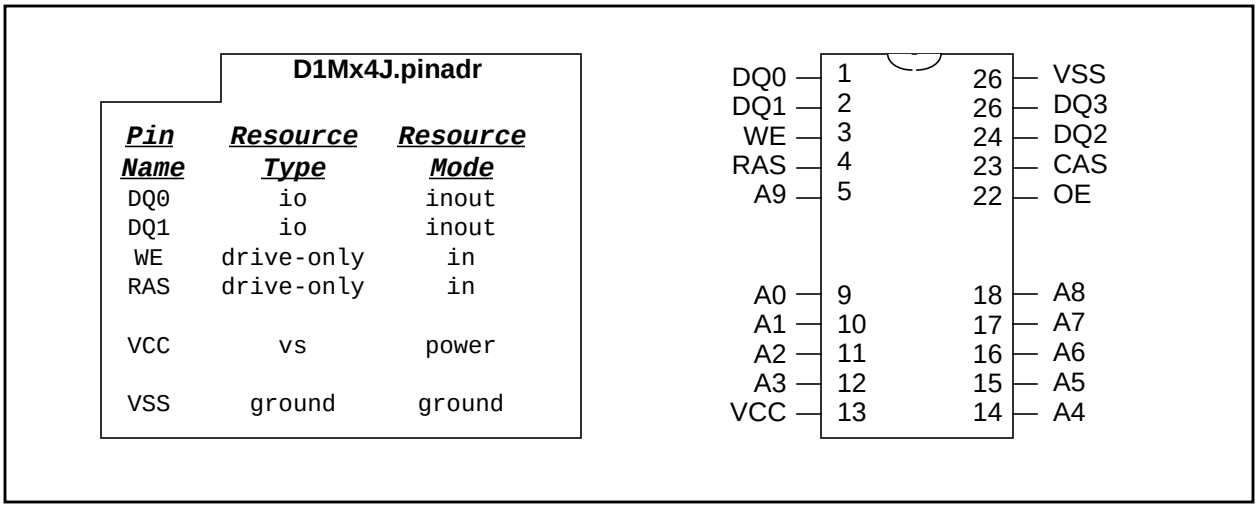
Chapter 1.2: Creating Pin Database Files

Objective: ***Create .pinadr and .pindb files***
Reference: ***ADR900 Reference Manual***

Introduction

The pin database file contains a list of individual device pins, pin groups that the test program will use, and the type of tester resource each pin is connected to. This file is used by other modules and databases which need to program tester hardware based on the pin names.

The pin database creates a symbol for each device pin and pingroup. These symbols are used by the waveform, level, patgen, datagen, src, and MUT databases in order to program the hardware by referencing device pins. The hardware resource type for each pin is also specified, so checking can be done by other database software to make sure correct programming is done for each pin.



D1Mx4J.pinadr

Pin Name	Resource Type	Resource Mode
DQ0	io	inout
DQ1	io	inout
WE	drive-only	in
RAS	drive-only	in
VCC	vs	power
VSS	ground	ground

Figure 1.2-4: Illustration of Pin Database for 1Mx4 DRAM SOJ

Another database (the channel database) will actually assign the device pin names to the tester channels. This is covered in the next chapter.

The pin database is usually associated with a certain device package style, as it also contains the physical package pin names and numbers. For this reason, a pin database may be desired for each package style a device will be tested in.

The pin database can be created using one of two methods:

- ❒ Create an ADR file (.pinadr) containing the pinset statement using a text editor such as vi or Sun's Textedit. Then translate the ADR file into binary database file (.pindb) using adr900.
- ❒ Use the ADRedit tool to both create and translate the ADR file. ADRedit can be used to modify an existing ADR file, start with the pinadr framework, or create a new file. ADRedit provides a button to perform the translation and move the cursor to the line in the file where translation errors are encountered.

Note: For the .pinadr file, the translator also generates a .h file that contains *import* statement references to the pin names**.** This file is included in source code (.t) modules that need to reference the pin symbols.

Contents of the Pinadr File

The pin database is defined with the ADR ***pinset*** statement, inside a .pinadr file. One pinset statement is allowed per pinadr file. Within the pinset statement, there is typically a pin-form clause for identifying individual pins, and a group-form clause for identifying pin groups. For more information regarding the syntax of ADR statements, see the *ADR900 Reference Manual*

Specific guidelines for the pin database contents:

- ❒ Use upper case for pin names, as this makes it easy to distinguish them in source code and other database modules.
- ❒ In the standard datalog formats, the test result field represents a hexadecimal bitmask of the pins that failed. A bitmask can be created only when the logical name of the failing pin(s) ends in a number. For example, I/0 16 would be represented as 0_0 10000fnR. However, in the expanded datalog formats, failing pins are listed for each site after the test result. A zero represents a pass, and a non-zero represents a fail.

Pinadr File Example

```
(:comment "d1Mx4pins.pinadr")

(pinset d1Mx4_pins :comment "1Mx4 DRAM SOJ Pin Names, Groups, and Types"
        (default :pref no :comment nil)
        (pin-form (:name  :comment                    :mode    :resource-type)
                (WE      "Write Enable"               in       drive-only)
                (RAS     "Row Addr Select"            in       drive-only)
                (CAS     "Column Addr Select"         in       drive-only)
                (OE      "Output Enable"              in       drive-only)

                (A0      "Address 0"                  in       drive-only)
                (A1      "Address 1"                  in       drive-only)
                (A2      "Address 2"                  in       drive-only)
                (A3      "Address 3"                  in       drive-only)
                (A4      "Address 4"                  in       drive-only)
                (A5      "Address 5"                  in       drive-only)
                (A6      "Address 6"                  in       drive-only)
                (A7      "Address 7"                  in       drive-only)
                (A8      "Address 8"                  in       drive-only)
                (A9      "Address 9"                  in       drive-only)

                (DQ0     "Data Input/Output 0"        inout    io)
                (DQ1     "Data Input/Output 1"        inout    io)
                (DQ2     "Data Input/Output 2"        inout    io)
                (DQ3     "Data Input/Output 3"        inout    io)

                (VCC     "+5V Power Pin"              power    vs)
                (VSS     "Vss/GND Pin"                ground   ground)
        )

        (default :mode inherit :pref no :comment nil)
        (group-form (:name          :elements)
                (CLKS               (WE RAS CAS OE))
                (ADDRS              (A0 A1 A2 A3 A4 A5 A6 A7 A8 A9))
                (INPUTS             (CLKS ADDRS))
                (IOS                (DQ0 DQ1 DQ2 DQ3))
                (DQS                (DQ0 DQ1 DQ2 DQ3))
                (OUTPUTS            (DQ0 DQ1 DQ2 DQ3))
                (ALL_PINS           (CLKS ADDRS IOS))
                (POWER_PINS         (VCC))
        )
)
```

Chapter 1.3: Creating Channel Database Files

Objective: ***Create .chanadr and .chandb files***
Reference: ***ADR900 Reference Manual,***
J990 Series Catch RAM Manual

Introduction

The channel database file contains a description of how the analog interface (interface board) is wired. This description includes the list of tester channels and voltage sources that are electrically connected to each device pin on each test site.

The channel database is stored in .chanadr files which contain chansets. If a single test program will be used with interface boards that are wired differently, a separate chanset statement would be created for each of the interfaces.

The channel database can be created using one of two methods:

- ❒ Create an ADR file (.chanadr) containing the chanset statement using a text editor, then translate the ADR file into binary database file (.chandb) using adr900.
- ❒ Use the ADRedit tool to both create and translate the ADR file.

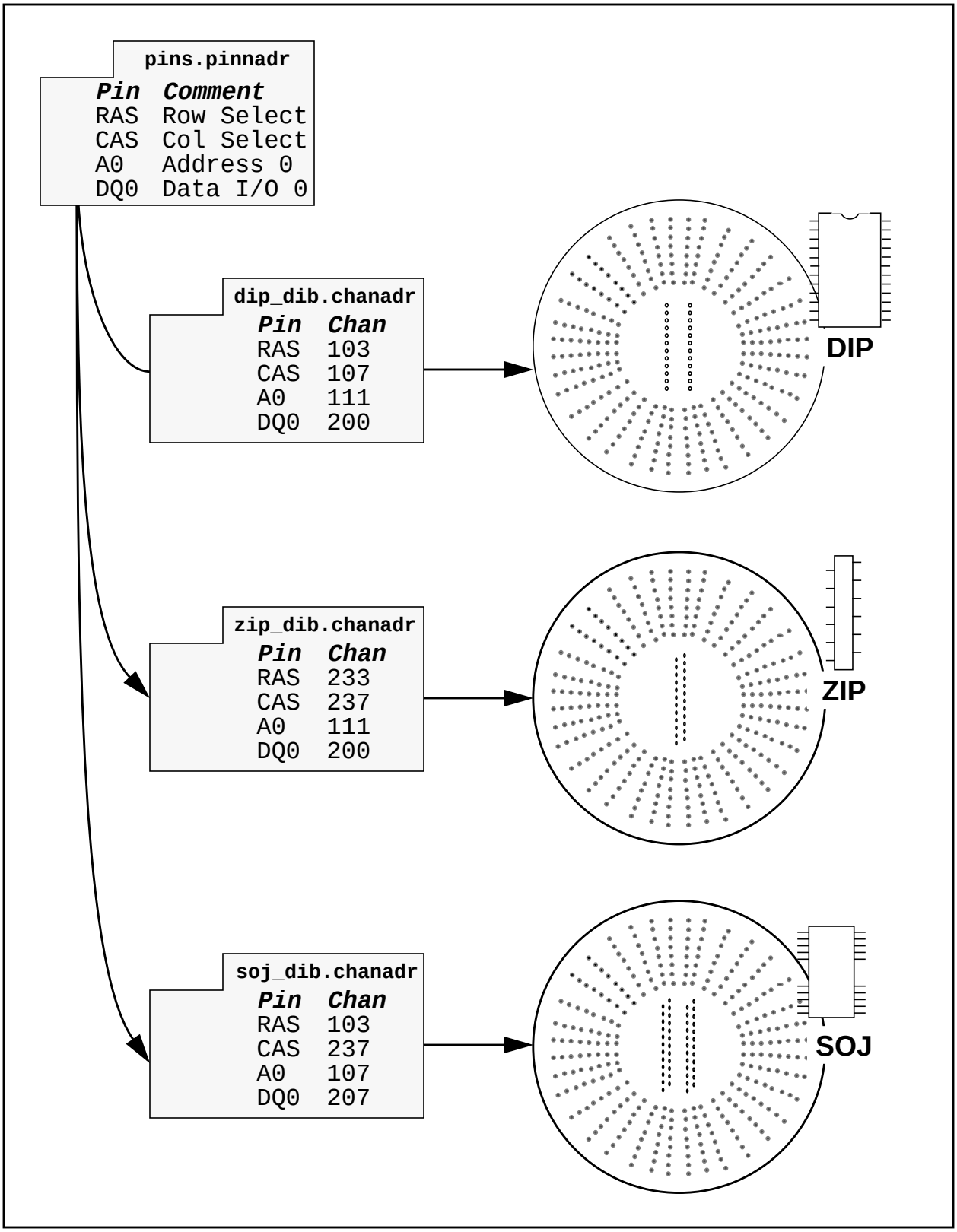


Figure 1.3-1 One Test Program Can be Used With Different Interfaces.

Selecting Channels for the Device Pins

Since there are two types of channels in a J990 Series test head, the programmer must select either a drive-only or an IO channel to be connected to the device pin. This decision should be based on whether there will be output data from the device that the tester needs to compare.

- ❒ Device output pins should be connected to IO channels.
- ❒ Device input pins (including data in pins) should be connected to drive-only channels.

The channels in the test head are referenced by both signal names and test head channel numbers. For example, IO channel 0 is also test head channel 200 on the J997.

In general, it is a good idea to select channels based on their signal name (Driver 0, site 0 or IO channel 12, etc.), instead of the channel number.

- ❒ Signal names reflect how the tester is organized. (IO channels 0 through 7 and Drivers 0 through 7 are the first channels stuffed into the system.)
- ❒ Channel numbers reflect a location on the interface board. (Not all systems will have all channel numbers available.)

In addition, there are guidelines for selecting each type of channel. There is a *Channel Selection Guide* available, which gives additional help and guidance when selecting appropriate channels.

Selecting IO Channels

IO channels are totally independent, as each has its own timing, format, and PMU. IO channels can be selected with a minimum set of guidelines.

The guidelines for selecting IO channels include:

- ❒ Start by selecting the list of IO channels to be used for the testing application. For example, if testing 8 devices, 1Mx4 DRAMs, then 32 IO channels would be needed. IO channels 0 through 31 would be a logical choice.
- ❒ IO channels should be assigned to each device pin to allow for the shortest possible path length to the pin. This will minimize capacitance at the device output, and reduce strobe restrictions relative to the HiZ/drive transitions.

Note: For more information about channel allocation and selection for Catch RAM, Redundancy, and Bitmap applications, refer to the **J990 Series Catch RAM Manual**, Chapter 3.1 Catch RAM, *DIB Wiring Considerations*.

Selecting Drive-Only Channels

Drive-only channels are associated with a hardware site. For example, a single drive-only channel card in the J997 has two signals fanned out to four sites (eight channels total). When cards are added to a J997 system, it can be expanded to provide additional functions (up to 40 functions), or it can be expanded to provide additional sites (up to 8 sites).

A fully stuffed J997 system can have 40 independent functions for eight sites. Alternatively, if only 20 functions are needed, then up to 16 sites can be used.

It is important to know how many drive-only channels are in a particular system and how it is stuffed before selecting drive-only channels for a test program.

The guidelines for selecting drive-only channels include:

- ❒ Start by selecting the list of drive-only channels to be used for the testing application. For example, if testing 8 devices, 1Mx4 DRAMs, then each site will need 14 drivers (10 address and 4 clocks). Since each driver signal is fanned out to four sites, 28 drive-only channels would be needed. ((28 drivers x 4 hardware sites) is equivalent to (14 driver signals x 8 device sites).)
- ❒ Drive-only channels should be assigned to each device pin to permit easy, sequential wiring of the channels. For example, if device pins 1 and 2 were to be connected to drive-only channels, it makes sense to connect these to drivers 0 and 1, since these channels are adjacent on the interface board and in the test head. Having a test head stuffing sheet available helps to choose signals that are adjacent in the test head.
- ❒ Select the sites of the drive-only channels to be consistently connected to the device site. For example, driver 0 — site 0 and driver 1— site 0 should go to the same device site. Driver 0 — site 1 and driver 1— site 1 should go to the same pins on a different site.

Channel Selection Programming

Once the list of signal names is assigned to the device pins, use the signal name to channel number tables (Tables 1 and 2 which follow) to determine which channel numbers to specify in the .chanadr file.

Use the tables at the end of this chapter to determine signal names from channel numbers and vice versa.

Keep in mind that channel numbers represent a location on the interface board, and do not correspond sequentially to the signal names.

Also not all channel numbers are used for a particular system type. For example, the J997 does not use channel numbers 16 through 23. These are unused locations on a J997 interface board.

When first writing a program, it may be useful to start with a hypothetical channel database, especially if an interface board is not yet available. This will enable program debugging before wiring an interface. When the interface wiring is known, the channel database can be updated with the correct information.

Channel Database Example: Quad Site 1Mx4 DRAM DIB

```
(:comment "d1Mx4j.chanadr")

; - under :name column is the name which will be used by this program
; - under the :site column is the channel number (DIB location) for the
;    particular channel that pin is wired to
; - under the :phys-name column is the package pin designator (number)
; - channels were chosen to help with coax cable placement
; - VSS (gnd) entry is shown just for documenting code, it serves no purpose

; The intended site locations on DIB illustrated as follows:
;
;           oct 0    oct 7
;   oct 1                     oct 6
;           site 2   site 3
;
;           site 0   site 1
;   oct 4                     oct 3
;           oct 5    oct 2

(chanset d1Mx4j_chans :comment "Channels for 1Mx4 DRAM SOJ DIB"
    (default :comment nil)
    (chan-form
           (:name   :site0   :site1   :site2   :site3   :phys-name) ; signal

           (DQ0     200      376      204      380      "Pin 1")
           (DQ1     201      377      205      381      "Pin 2")
           (WE      140      141      142      143      "Pin 3")    ; D5
           (RAS     132      133      134      135      "Pin 4")    ; D1
           (A9      192      193      194      195      "Pin 5")    ; D2

           (A0      212      213      214      215      "Pin 9")    ; D7
           (A1      152      153      154      155      "Pin 10")   ; D8
           (A2      156      157      158      159      "Pin 11")   ; D9
           (A3      208      209      210      211      "Pin 12")   ; D6
           (VCC     0        1        2        3        "Pin 13")

           (A4      368      369      370      371      "Pin 14")   ; D12
           (A5      136      137      138      139      "Pin 15")   ; D4
           (A6      312      313      314      315      "Pin 16")   ; D10
           (A7      196      197      198      199      "Pin 17")   ; D3
           (A8      372      373      374      375      "Pin 18")   ; D13

           (OE      316      317      318      319      "Pin 22")   ; D11
           (CAS     128      129      130      131      "Pin 23")   ; D0
           (DQ2     202      378      206      382      "Pin 24")
           (DQ3     203      379      207      383      "Pin 25")
;          (VSS     gnd      gnd      gnd      gnd      "Pin 26")
    )
)
```

Table 1.3-1: J997 IO Channel Numbers According to Signal Names

Signal	***Chan***	***Signal***	***Chan***	***Signal***	***Chan***	***Signal***	***Chan***
io0:	200	io18:	218	io36:	252	io54:	238
io1:	201	io19:	219	io37:	253	io55:	239
io2:	202	io20:	220	io38:	254	io56:	344
io3:	203	io21:	221	io39:	255	io57:	345
io4:	204	io22:	222	io40:	328	io58:	346
io5:	205	io23:	223	io41:	329	io59:	347
io6:	206	io24:	360	io42:	330	io60:	348
io7:	207	io25:	361	io43:	331	io61:	349
io8:	376	io26:	362	io44:	332	io62:	350
io9:	377	io27:	363	io45:	333	io63:	351
io10:	378	io28:	364	io46:	334	io64:	272
io11:	379	io29:	365	io47:	335	io65:	273
io12:	380	io30:	366	io48:	232	io66:	274
io13:	381	io31:	367	io49:	233	io67:	275
io14:	382	io32:	248	io50:	234	io68:	276
io15:	383	io33:	249	io51:	235	io69:	277
io16:	216	io34:	250	io52:	236	io70:	278
io17:	217	io35:	251	io53:	237	io71:	279

Table 1.3-2: J993, J995, J997 Driver Channel Numbers According to Site/Signal Names

MCS Card	Signal	Slot #	Site 0	Site 1	Site 2	Site 3	Site 4	Site 5	Site 6	Site 7	Slot #
MCS 0 Slot #13	d0:	P30	128	129	130	131	56	57	58	59	P41
	d1:		132	133	134	135	60	61	62	63	
	d2:	P29	192	193	194	195	112	113	114	115	P42
	d3:		196	197	198	199	116	117	118	119	
	d4:	P28	136	137	138	139	40	41	42	43	P46
	d5:		140	141	142	143	44	45	46	47	
	d6:	P24	208	209	210	211	96	97	98	99	P47
	d7:		212	213	214	215	100	101	102	103	
MCS 1 Slot #14	d8:	P23	152	153	154	155	32	33	34	35	P48
	d9:		156	157	158	159	36	37	38	39	
	d10:	P50	312	313	314	315	384	385	386	387	P39
	d11:		316	317	318	319	388	389	390	391	
	d12:	P51	368	369	370	371	448	449	450	451	P38
	d13:		372	373	374	375	452	453	454	455	
	d14:	P55	296	297	298	299	392	393	394	395	P37
	d15:		300	301	302	303	396	397	398	399	
MCS 2 Slot #15	d16:	P56	352	353	354	355	464	465	466	467	P33
	d17:		356	357	358	359	468	469	470	471	
	d18:	P67	288	289	290	291	408	409	410	411	P32
	d19:		292	293	294	295	412	413	414	415	
	d20:	P17	256	257	258	259	440	441	442	443	P72
	d21:		260	261	262	263	444	445	446	447	
	d22:	P16	320	321	322	323	496	497	498	499	P73
	d23:		324	325	326	327	500	501	502	503	
MCS 3 Slot #16	d24:	P15	264	265	266	267	424	425	426	427	P77
	d25:		268	269	270	271	428	429	430	431	
	d26:	P11	336	337	338	339	480	481	482	483	P78
	d27:		340	341	342	343	484	485	486	487	
	d28:	P10	280	281	282	283	416	417	418	419	P79
	d29:		284	285	286	287	420	421	422	423	
	d30:	P63	184	185	186	187	0	1	2	3	P8
	d31:		188	189	190	191	4	5	6	7	
MCS 4 Slot #17	d32:	P64	240	241	242	243	64	65	66	67	P7
	d33:		244	245	246	247	68	69	70	71	
	d34:	P68	168	169	170	171	8	9	10	11	P6
	d35:		172	173	174	175	12	13	14	15	
	d36:	P69	224	225	226	227	80	81	82	83	P2
	d37:		228	229	230	231	84	85	86	87	
	d38:	P70	160	161	162	163	24	25	26	27	P1
	d39:		164	165	166	167	28	29	30	31	

Table 1.3-3: J997 Signal Names According to Channel Numbers

Chan	Sig
0	d30s4
1	d30s5
2	d30s6
3	d30s7
4	d31s4
5	d31s5
6	d31s6
7	d31s7
8	d34s4
9	d34s5
10	d34s6
11	d34s7
12	d35s4
13	d35s5
14	d35s6
15	d35s7
...	
...	
24	d38s4
25	d38s5
26	d38s6
27	d38s7
28	d39s4
29	d39s5
30	d39s6
31	d39s7
32	d8s4
33	d8s5
34	d8s6
35	d8s7
36	d9s4
37	d9s5
38	d9s6
39	d9s7
40	d4s4
41	d4s5
42	d4s6
43	d4s7
44	d5s4
45	d5s5
46	d5s6
47	d5s7
...	
...	

Chan	Sig
56	d0s4
57	d0s5
58	d0s6
59	d0s7
60	d1s4
61	d1s5
62	d1s6
63	d1s7
64	d32s4
65	d32s5
66	d32s6
67	d32s7
68	d33s4
69	d33s5
70	d33s6
71	d33s7
...	
...	
80	d36s4
81	d36s5
82	d36s6
83	d36s7
84	d37s4
85	d37s5
86	d37s6
87	d37s7
...	
...	
96	d6s4
97	d6s5
98	d6s6
99	d6s7
100	d7s4
101	d7s5
102	d7s6
103	d7s7
...	
...	
112	d2s4
113	d2s5
114	d2s6
115	d2s7

Chan	Sig
116	d3s4
117	d3s5
118	d3s6
119	d3s7
...	
...	
128	d0s0
129	d0s1
130	d0s2
131	d0s3
132	d1s0
133	d1s1
134	d1s2
135	d1s3
136	d4s0
137	d4s1
138	d4s2
139	d4s3
140	d5s0
141	d5s1
142	d5s2
143	d5s3
...	
...	
152	d8s0
153	d8s1
154	d8s2
155	d8s3
156	d9s0
157	d9s1
158	d9s2
159	d9s3
160	d38s0
161	d38s1
162	d38s2
163	d38s3
164	d39s0
165	d39s1
166	d39s2
167	d39s3
168	d34s0
169	d34s1
170	d34s2
171	d34s3

Chan	Sig
172	d35s0
173	d35s1
174	d35s2
175	d35s3
...	
...	
184	d30s0
185	d30s1
186	d30s2
187	d30s3
188	d31s0
189	d31s1
190	d31s2
191	d31s3
192	d2s0
193	d2s1
194	d2s2
195	d2s3
196	d3s0
197	d3s1
198	d3s2
199	d3s3
200	io0
201	io1
202	io2
203	io3
204	io4
205	io5
206	io6
207	io7
208	d6s0
209	d6s1
210	d6s2
211	d6s3
212	d7s0
213	d7s1
214	d7s2
215	d7s3
216	io16
217	io17
218	io18
219	io19
220	io20
221	io21

Chan	Sig
222	io22
223	io23
224	d36s0
225	d36s1
226	d36s2
227	d36s3
228	d37s0
229	d37s1
230	d37s2
231	d37s3
232	io48
233	io49
234	io50
235	io51
236	io52
237	io53
238	io54
239	io55
240	d32s0
241	d32s1
242	d32s2
243	d32s3
244	d33s0
245	d33s1
246	d33s2
247	d33s3
248	io32
249	io33
250	io34
251	io35
252	io36
253	io37
254	io38
255	io39
256	d20s0
257	d20s1
258	d20s2
259	d20s3
260	d21s0
261	d21s1
262	d21s2
263	d21s3

Table 1.3-3: J997 Signal Names According to Channel Numbers (continued)

J994 Channel Number to Signal Names and Signal Name to Channel Number Charts (7 pages)

Chan	Sig
264	d24s
265	d24s
266	d24s
267	d24s3
268	d25s0
269	d25s1
270	d25s2
271	d25s3
272	io64
273	io65
274	io66
275	io67
276	io68
277	io69
278	io70
279	io71
280	d28s0
281	d28s1
282	d28s2
283	d28s3
284	d29s0
285	d29s1
286	d29s2
287	d29s3
288	d18s0
289	d18s1
290	d18s2
291	d18s3
292	d19s0
293	d19s1
294	d19s2
295	d19s3
296	d14s0
297	d14s1
298	d14s2
299	d14s3
300	d15s0
301	d15s1
302	d15s2
303	d15s3
...	
...	

Chan	Sig
313	d10s1
314	d10s2
315	d10s3
316	d11s0
317	d11s1
318	d11s2
319	d11s3
320	d22s0
321	d22s1
322	d22s2
323	d22s3
324	d23s0
325	d23s1
326	d23s2
327	d23s3
328	io40
329	io41
330	io42
331	io43
332	io44
333	io45
334	io46
335	io47
336	d26s0
337	d26s1
338	d26s2
339	d26s3
340	d27s0
341	d27s1
342	d27s2
343	d27s3
344	io56
345	io57
346	io58
347	io59
348	io60
349	io61
350	io62
351	io63

Chan	Sig
355	d16s3
356	d17s0
357	d17s1
358	d17s2
359	d17s3
360	io24
361	io25
362	io26
363	io27
364	io28
365	io29
366	io30
367	io31
368	d12s0
369	d12s1
370	d12s2
371	d12s3
372	d13s0
373	d13s1
374	d13s2
375	d13s3
376	io8
377	io9
378	io10
379	io11
380	io12
381	io13
382	io14
383	io15
384	d10s4
385	d10s5
386	d10s6
387	d10s7
388	d11s4
389	d11s5
390	d11s6
391	d11s7
392	d14s4
393	d14s5
394	d14s6
395	d14s7

Chan	Sig
399	d15s7
...	
...	
408	d18s4
409	d18s5
410	d18s6
411	d18s7
412	d19s4
413	d19s5
414	d19s6
415	d19s7
416	d28s4
417	d28s5
418	d28s6
419	d28s7
420	d29s4
421	d29s5
422	d29s6
423	d29s7
424	d24s4
425	d24s5
426	d24s6
427	d24s7
428	d25s4
429	d25s5
430	d25s6
431	d25s7
...	
...	
440	d20s4
441	d20s5
442	d20s6
443	d20s7
444	d21s4
445	d21s5
446	d21s6
447	d21s7
448	d12s4
449	d12s5
450	d12s6
451	d12s7

Chan	Sig
452	d13s4
453	d13s5
454	d13s6
455	d13s7
...	
...	
464	d16s4
465	d16s5
466	d16s6
467	d16s7
468	d17s4
469	d17s5
470	d17s6
471	d17s7
...	
...	
480	d26s4
481	d26s5
482	d26s6
483	d26s7
484	d27s4
485	d27s5
486	d27s6
487	d27s7
...	
...	
496	d22s4
497	d22s5
498	d22s6
499	d22s7
500	d23s4
501	d23s5
502	d23s6
503	d23s7

Chapter 1.4: Creating MUT Database Files

Objective: ***Create .mutadr and .mutdb files***
Reference: ***ADR900 Reference Manual***

Introduction

The MUT database file contains a description of the memory-under-test. This description includes the device size, shape, topology (internal layout), address scrambling, topological data inversion, and defines what information is to appear on each device pin.

The MUT database is designed so that the programmer can describe the device in familiar terms that are not necessarily specific to the hardware. Typically a programmer can create the MUT description using a device block diagram, a diagram of the memory array layout (topology), and an address scrambling scheme.

The MUT database is used by the job exec to program the address scrambler, and select specific address bits to drive the address pins on the device.

The MUT database can be created using one of two methods:

- ❒ Create an ADR file (.mutadr) containing the mutset statement using a text editor, then translate the ADR file into binary database file (.mutdb) using adr900.
- ❒ Use the ADRedit tool to both create and translate the ADR file.

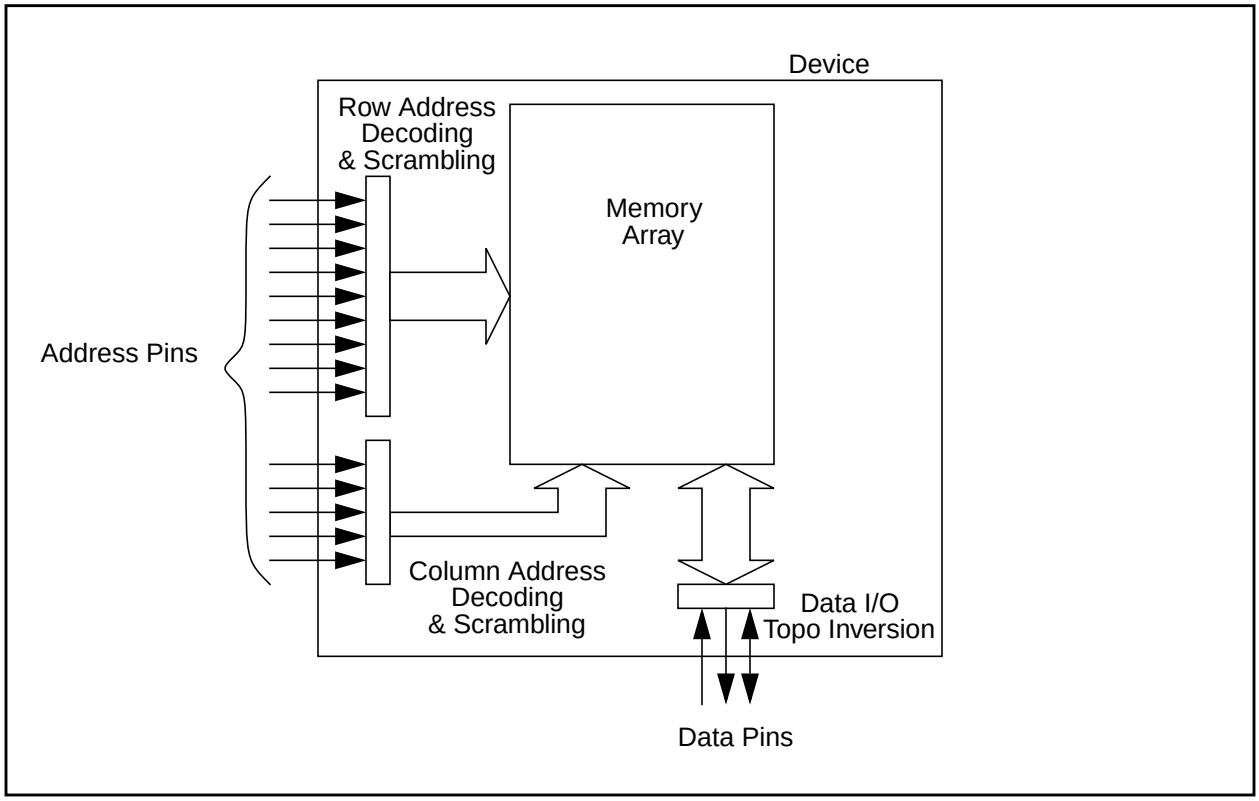


Figure 4.1-1 Illustration of Memory Array as Perceived by MUT Database

As the preceding figure illustrates, the MUT database perceives the device as a memory array that is accessed by address pins that get decoded and scrambled. The data that is stored in the array depends on the topological inversion logic.

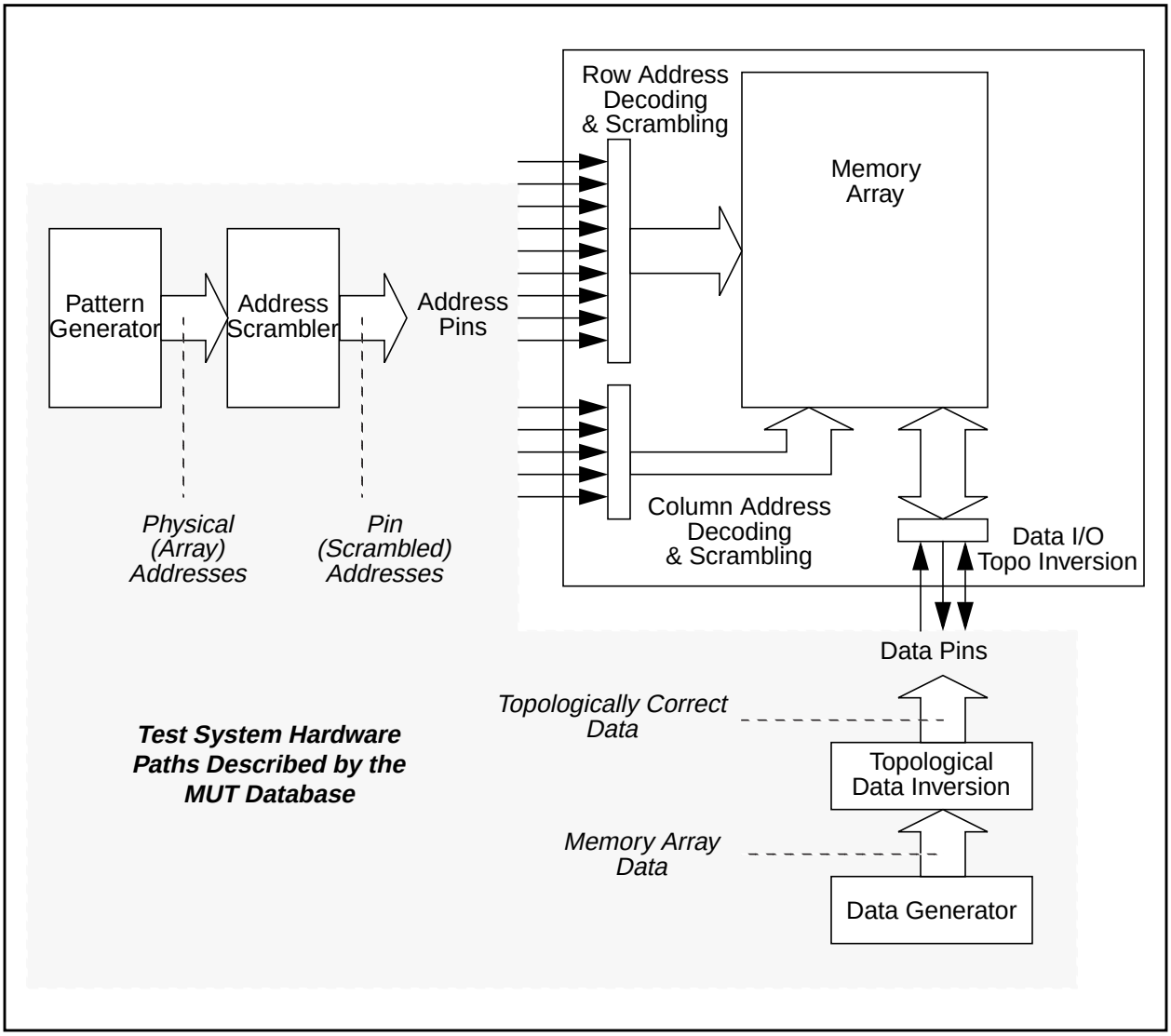


Figure 1.4-2 Illustration of Memory Array Testing as Perceived by MUT Database

When the device is tested, the tester must be set up to accurately test the memory array. The patgen generates the address sequences for the memory array. These are scrambled for the device pins, so that when the device address decoding and scrambling is done, the address produced by the patgen is the same as that accessed in the memory array. The same technique is done for topologically inverted data that is stored in the array.

The gray area in the figure represents the test system hardware that is set up based on the contents of the MUT database.

Definition of MUT Terminology

The following set of terms are used in this manual with respect to memory devices.

row or column These are used to describe the axis of a device. They are not tied to the horizontal and vertical axis of the display. They are used as modifiers when describing addresses or coordinates.

coordinate or location Locations on a display that correspond to memory cells. The terms *row coordinates* or *column coordinates* can be used to reference the coordinates or locations along the row or column axis. Other coordinate terms include: *device coordinates*, *array coordinates*, and *RA region coordinates*.

physical address Row or column address that corresponds to a bit or bits within an array. These are generated by the patgen. These can be considered the *internal* address.

scrambled address Row or column address that are presented to the device pins. These are generated by the *address scrambler*. These can be considered the *external* address.

topology General term that is used to refer to the layout of the device, usually as it applies to the display.

address scrambling The act of converting physical addresses (internal or patgen addresses) to scrambled addresses (device pin addresses). The is done in hardware by the *address scrambler*.

Address Identifiers

Teradyne has devised certain address identifiers which will be used when actually writing a mutadr file.

pr0 .. prN are identifiers for physical row address bits. There is a system-defined relationship between these identifiers and the X address bits generated by the patgen (i.e., x0 = pr0, etc.). These are the non-scrambled row address bits.

pc0 .. pcN are identifiers for physical column address bits. There is a system-defined relationship between these identifiers and the Y address bits generated by the patgen (i.e., y0 = pc0, etc.). These are the non-scrambled column address bits.

Contents of the Mutadr File

The MUT database is defined with the ADR ***mutset*** statement, inside a .mutadr file. Multiple mutset statements are allowed per mutadr file.

The (use pinset) statement must appear before any mutset statements in the file.

For more information regarding the syntax of ADR statements, see the *ADR900 Reference Manual.*

Mutset Contents

The MUT description is contained in a mutset statement, which has different sections or clauses. The clauses within the mutset are:

- ❒ size - to describe the number of addresses and data width
- ❒ shape - to describe the number of address rows and columns
- ❒ address - to describe the information to appear at the address pins
- ❒ data - to describe the information to appear at the data pins

size

The size is expressed with the :words argument to specify the number of addresses, and the :width argument for number of data bits.

```
(size  :words 131072  :width 8)    ; 128Kx8 device
```

shape

The shape is expressed with the :rows and :columns arguments. These numbers describe the number of address rows and columns.

```
(shape  :rows 512  :columns 256)
```

address

The address clause includes either the map-function-form or the map-value-form table to specify which pins are address pins and how the address scrambler is to be programmed. In addition, it also defines whether or not address muxing is on.

map-function-form

This table describes the mapping of pins to physical address bits (rows/columns) using boolean expressions to define the address scrambling. Typically, each pin's address will be described by giving a boolean expression containing physical address bit identifiers.

```
(address
        (map-function-form
                (:pin    :physical)
                (A0      "pr0 ^ pr1")
                (A1      "~(pr0 ^ pr1)")
                (A2      "pr2")
                (A3      "pc0")
                (A4      "pc1")
        )
)
```

A one-to-one mapping of physical to scrambled address bits to the device pins is expressed by just mapping the physical address bits to the pins:

```
(address
        (map-function-form
                (:physical  :pin)
                (pr0        A0)
                (pr1        A1)
                (pr2        A2)
                (pr3        A3)
                (pr4        A4)
                (pr5        A5)
                (pr6        A6)
                (pr7        A7)

                (pc0        A8)
                (pc1        A9)
                (pc2        A10)
                (pc3        A11)
                (pc4        A12)
                (pc5        A13)
                (pc6        A14)
        )
)
```

map-value-form

This table defines address scrambling by explicitly listing values that are to appear at the pins. The user lists the address pins and scrambled pin values under the :pin header, the physical address identifiers and values in sequence under the :physical header, and the resultant scrambled pin address values.

Pin names and address identifiers are always listed in MSB-to-LSB order for binary decoding of a value list.

```
(address
   (map-value-form         ; row address scrambling
         (:physical                :pin)
         (pr3 pr2 pr1 pr0)         (A3 A2 A1 A0)
         (0..15)                   (15 13 11 9 7 5 3 1
                                   0 2 4 6 8 10 12 14)
   )
   (map-value-form         ; col address scrambling
         (:physical                :pin)
         (pc3 pc2 pc1 pc0)         (A7 A6 A5 A4)
         (0..15)                   (0 2 4 6 8 10 12 14
                                   15 13 11 9 7 5 3 1)
   )
)
```

data

The data clause contains a map-data-form table to define the topological data inversion on a per-pin basis. This clause is required, and must include all pins that will receive data from the data generator.

map-data-form

This table defines the data inversion (topological invert), and multiplexing of data information. through the use of an expression (function) for the device. Data is inverted when the quoted expression is true. This expression is a function of physical row and column bits. If no data inversion is needed on the data pins, "0" or nil must be specified for the pin.

```
; checkerboard data invert based on phys row bit 9 and col bit 8
(data
         (map-data-form
                  (:pin    :physical)
                  (DQS     "pr9 ^ pc8")
         )
)
```

In addition, this statement is needed to tell the job exec which pins are data pins. The function is a quoted string and evaluated by the database in the same manner as equations are handled.

Mutadr File Example - 32Kx8 SRAM

```
(use pinset "pins_s32Kx8.pindb" pinset_s32Kx8)

; address scrambling with no topo invert

(mutset mut_s32Kx8
         (size    :words 32768  :width 8)

         (shape   :rows 256  :columns 128)

         (address
                  (map-function-form
                           (:physical  :pin)
                           (pr0        A12)
                           (pr1        A7)
                           (pr2        A6)
                           (pr3        A5)
                           (pr4        A0)
                           (pr5        A10)
                           (pr6        A9)
                           (pr7        A11)

                           (pc0        A8)
                           (pc1        A13)
                           (pc2        A14)
                           (pc3        A4)
                           (pc4        A3)
                           (pc5        A2)
                           (pc6        A1)
                  )
         )

         (data
                  (map-data-form
                           (:physical  :pin)
                           ("0"        (DQS))
                  )
         )
)
```

Pin Suffixes For Specifying Multiplexed Address or Data

The address identifiers will be used to specify the address information to appear at the pin. In the case of address multiplexed devices, two pieces of address information will need to appear at the pin. A pin suffix notation is used to identify the two halves of the address pin cycle.

This notation is PINNAME$SUFFIX:

- ❒ where PINNAME is a pin defined in a pinset, and suffix is 0 or 1.
- ❒ SUFFIX '0' means which data will be presented to the device "first".
- ❒ SUFFIX '1' means which data will be presented to the device "second".
- ❒ where "first" is defined as "xgate" or "d0-d1-d2" time, and "second" means the "ygate" or "d3-d4-d5" time, as defined by the formatter-mux hardware and the waveform database.

For example, when address multiplexing, two address bits are identified for each address pin:

```
(address
        (map-function-form
                (:pin    :physical)

                (A0$0    pr0)
                (A0$1    pc0)

                (A1$0    pr1)
                (A1$1    pc1)
        )
)
```

When multiplexing both address and data onto a pin, pin suffixes must be used in both the address clause and the data clause:

```
(address
        (map-function-form
                (:pin    :physical)
                (AD0$0   pr0)
                (AD1$0   pc0)
        )
)
(data
        (map-data-form
                (:pin    :physical)
                (AD0$1   nil)
                (AD1$1   nil)
        )
)
```

For pins that are dual purpose (sometimes address, sometimes data), pin suffixing is typically not used, since only one bit of data is needed per cycle.

Mutadr File Example - Address Multiplexed 1Mx4 DRAM

```
; MUT Database for 1Mx4 DRAM Device - 10 Address Pins, 4 Data I/O Pins
; Address multiplexed inputs, no scrambling or topo inversion

(use pinset "d1Mx4pins.pindb" d1Mx4_pins)

(mutset d1Mx4
         (size :words 1048576 :width 4)
         (shape :rows 1024 :columns 1024)

         (address
           (map-function-form
                  (:pin    :physical)

                  (A0$0    pr0)
                  (A0$1    pc0)
                  (A1$0    pr1)
                  (A1$1    pc1)
                  (A2$0    pr2)
                  (A2$1    pc2)
                  (A3$0    pr3)
                  (A3$1    pc3)
                  (A4$0    pr4)
                  (A4$1    pc4)
                  (A5$0    pr5)
                  (A5$1    pc5)
                  (A6$0    pr6)
                  (A6$1    pc6)
                  (A7$0    pr7)
                  (A7$1    pc7)
                  (A8$0    pr8)
                  (A8$1    pc8)
                  (A9$0    pr9)
                  (A9$1    pc9)
           )
         )

         (data
           (map-data-form
                  (:pin    :physical)
                  (DQS     nil)
           )
         )
)
```

Chapter 1.5: Creating Specification Database Files

Objective: ***Create .specadr and .specdb files***
Reference: ***ADR900 Reference Manual***

Introduction

An important part of device-oriented programming is the specification of device specifications and parameters. These device specifications (specs) are taken directly from a device datasheet and entered into specification databases. The specification databases are later used with level and waveform equations to define how the hardware is to be programmed.

In general, timing (AC) and level (DC) specifications are placed in separate database files. Each file can contain any number of sets of specification values called ***specsets***. For example, a timing database file may have a specset of minimum specification values for a functional test, another specset for maximum values, etc.

Spec limits, Shmoo axis ranges, and adjust-until parameters can also be defined in the specification databases, but it is primarily the value of a specification that is used by the software. The value of the specification is used to manipulate the hardware and is also varied by the Shmoo and Specsheet tools.

The specification databases can be created using one of four methods:

- ❒ Create an ADR file (.specadr) containing specset statements using a text editor, then translate the ADR file into binary database file (.specdb) using adr900.
- ❒ Use the ADRedit tool to both create and translate the ADR file.
- ❒ Use the Specsheet tool to create binary database files (.specdb).
- ❒ Use the Specsheet tool to create an ADR database file (.specadr) and manually translate it to create the binary.

Either of the first two methods is recommended for beginning programmers.

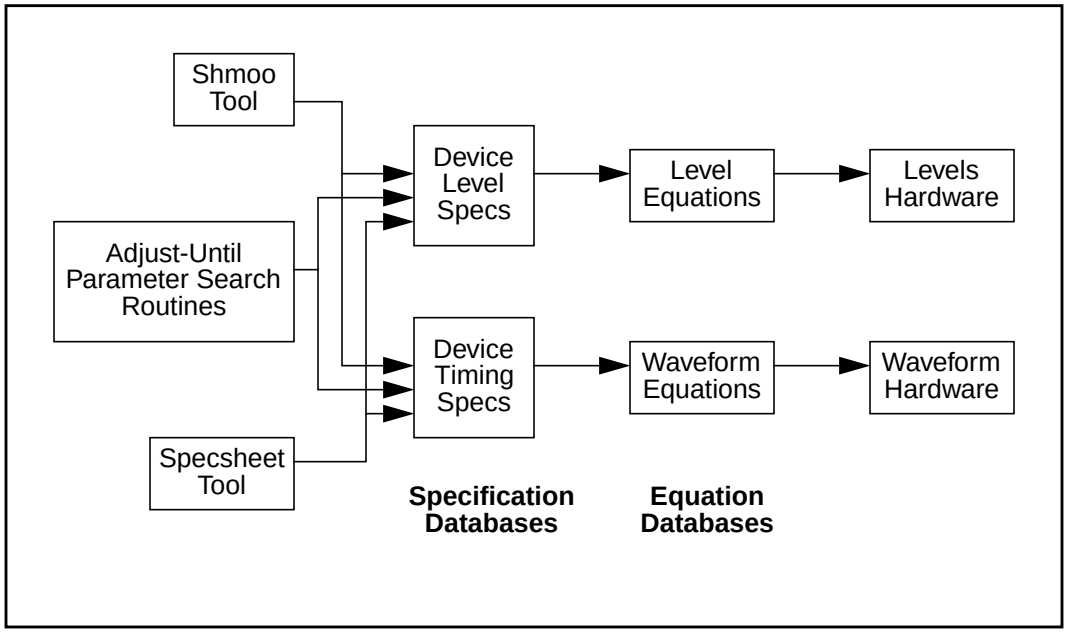


Figure 1.5-3 Relationship of Specifications to Hardware and Software

Contents of the Specadr File

The Spec database is defined with the ADR ***specs*** and ***specset*** statements, inside a .specadr file. Multiple ***specs*** and ***specset*** statements are allowed per file.

specs *statement*

The ***specs*** statement effectively *declares* the specifications. This declaration consists of the name (***:symbol***), description string (***:description***), units (***:units***), and printing format string (***:format***). Other attributes of the spec include a text label (***:item***), device spec identifier (***:devspec***), and a comment.

```
(specs
      (default :format "6.1" :units "nS"
        :devspec yes :comment nil :item nil)
      (spec-form
            (:symbol    :description)
            (tRC        "Random Read or Write Cycle Time")
            (tRAC       "Access time from RAS")
            (tCAC       "Access Time from CAS")
            (tOE        "Output Enable Time")
            (tAA        "Access Time from Column Address")
            (tRAS       "RAS Pulse Width")
            (tCAS       "CAS Pulse Width")
            (tRCD_min   "Min RAS to CAS Delay Time")
            (tRCD_max   "Max RAS to CAS Delay Time")
            (tT         "Driver Transition Time" :devspec no)
            (tHIZ       "Drive-HiZ Transition"  :devspec no)
      )
)
```

A spec must be first declared in a ***specs*** statement before being *defined* in a ***specset*** statement in a ***.specadr*** file.

specset *statement*

The *definition* of a spec and its associated values is done by the ***specset*** statement. A spec is typically assigned a value (***:value***), and other optional parameters depending on how the spec will be used. The value is used with the equations databases (levels and waveforms) to define a hardware value. The optional parameters include:

- ❒ Shmoo axis parameters (***:start*** and ***:stop***)
- ❒ spec limits (***:min*** and ***:max***)
- ❒ spec search and log (adjust-until) parameters (***:from***, ***:to***, ***:resolution***, ***:maxloops***, etc.)

When first creating a program, it is recommended to just program the value of a spec, as this reduces the program complexity and development time. When the other value parameters associated with a spec need to be used, they can be added.

```
(specset timing_nominal
      :comment "Nominal Timing for Basic Functional Tests"
      (default :start nil :stop nil :from nil :to nil
        :ratio 0 :min 0 :max 0 :seq-order 0 :backoff 0
        :result nil :first fail :resolution 0.1
        :log-setup 0 :method successive :maxloops 25)
      (set-form
            (:symbol :value)
            (tRC        200)
            (tRAC       120)
            (tCAC       30)
            (tOE        30)
            (tAA        60)
            (tRAS       130)
            (tCAS       50)
            (tRCD_min   25)
            (tRCD_max   50)
            (tT         3)
            (tHIZ       5)
      )
)
```

Level specs and timing specs are usually contained in separate files. It is recommended that level spec file names contain **levels** and timing spec file names contain **timing**, such as **levels.specadr** and **timing.specadr.**

Specification Database Example: Level Specs for DRAM

```
(specs
     (default :format "6.1" :units "V" :devspec yes :comment nil :item nil)
     (spec-form
            (:symbol :description)
            (Vcc     "Supply Voltage")
            (Vih     "Input High Voltage")
            (Vil     "Input Low Voltage")
            (Voh     "Output High Voltage")
            (Vol     "Output Low Voltage")
     )
     (default :format "6.0" :units "mA" :devspec yes :comment nil :item nil)
     (spec-form
            (:symbol :description)
            (Ioh     "Output High Current":format "6.1")
            (Iol     "Output Low Current":format "6.1")
            (Icc     "Operating Current (Random Read/Write)")
            (Iccs    "Standby Current")
            (Icclp   "Low Power Standby Current")
            (Iccr    "RAS only Refresh Current")
            (Iccp    "Page Mode Operating Current")
     )
     (default :format "6.3" :units "uA" :devspec yes :comment nil :item nil)
     (spec-form
            (:symbol :description)
            (Ii      "Input Leakage Current")
            (Io      "Output Leakage Current")
     )
)

(specset nom_levs :comment "Nominal Levels for Functional Tests"
     (default :start 0 :stop 0 :from 0 :to 0 :ratio 0 :seq-order 0
            :min nil :max nil :backoff 0 :result nil :first fail
            :resolution 0.5 :log-setup 0 :method successive :maxloops 25)
     (set-form
            (:symbol :value)
            (Vcc     5.0)
            (Vih     3.0)
            (Vil     0.0)
            (Voh     2.0)
            (Vol     0.8)
            (Ioh     -3.0)
            (Iol     2.0)
            (Icc     80)
            (Iccs    2)
            (Icclp   1)
            (Iccr    80)
            (Iccp    60)
            (Ii      0)
            (Io      0)
     )
)
```

Chapter 1.6: Creating Level Database Files

Objective: ***Create .leveladr and .leveldb files***
Reference: ***ADR900 Reference Manual***

Introduction

In order to manipulate the hardware based on device specifications, a set of equations is used. These equations calculate test system values based on data from the specification databases. These equations are put into level databases (for calculating pin and voltage source levels) and waveform databases (for calculating timing values). This chapter discusses the level databases.

Each pin or pin group can have multiple level values that need to be programmed, for example Vil and Vih. Each value for each pin symbol can have an equation.

The level databases can be created using one of four methods:

- Create an ADR file (.leveladr) containing ***pins*** and ***eqset*** statements using a text editor, then translate the ADR file into binary database file (.leveldb) using the ADR900 translator.
- Use the ADRedit tool to both create and translate the ADR file.
- Use the Calcsheet tool to create binary database files (.leveldb).
- Use the Calcsheet tool to create an ADR database file (.leveladr) and manually translate it to create the binary.

Either of the first two methods is recommended for beginning programmers.

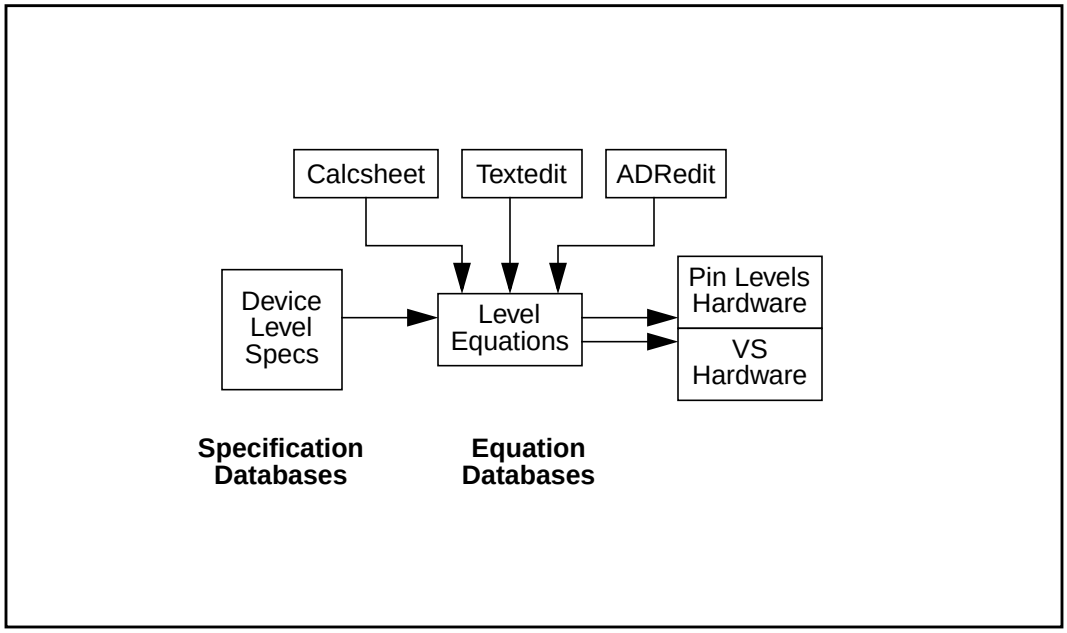


Figure 1.6-1 Relationship of Levels Equations to Hardware and Software

Contents of the Leveladr File

The level database is defined with the ADR ***pins*** and ***eqset*** statements, inside a .leveladr file. Only one ***pins*** and one ***eqset*** statement are allowed per file. The ***use pinset*** and ***range*** statements must appear before the level definitions.

use pinset statement

The ***use pinset*** statement effectively *declares* the pins that can be manipulated by the database file. This declaration must appear before any other statements in the file.

range statement

The ***range*** statement defines the pin driver level range that is to be in effect for this level set. One of four level ranges can be selected for the drivers. Since it takes 750 ms to alter the levels range, changing ranges too frequently or using different ranges throughout your leveladr file will slow down your program.

pins statement

The ***pins*** statement defines the sequence to be used when manipulating the pin levels. The sequence is executed in increasing order when the VS level is increasing (or power is being turned on), and in decreasing order when the VS levels is decreasing. Typically, the power pin on the device should be programmed with sequence 1, and driver pins programmed with a higher sequence number. Extensive information on power sequencing can be found in the **J990 Series ADR Reference Manual** in the *Pins Statement* description.

eqset statement

The ***eqset*** statement contains all the level equations for each pin on the device. Drive-only and IO channels are programmed with the ***equation-form*** clause, and voltage source pins are programmed with the ***vs-form*** clause.

Within the ***equation-form***, drive-only pins have only two levels values:

- **vih**: driver high voltage
- **vil**: driver low voltage

IO pins have both driver and comparator levels:

- **vih**: driver high voltage
- **vil**: driver low voltage
- **voh**: comparator high voltage
- **vol**: comparator low voltage
- **vch**: comparator clamp high voltage
- **vcl**: comparator clamp low voltage
- **vt**: HiZ termination voltage
- **ioh**: current load for logic high
- **iol**: current load for logic low

Both ioh and iol should be programmed to be positive values, as this is how the hardware values are defined.

Within the ***vs-form***, both main and alternate voltage levels are programmed, and the current limit (current clamp) specified.

The equations for these hardware values can be any valid C expression. These expressions are evaluated (and values calculated) at program load time by the database loader. The specifications used with the equations are determined by the workspace. This allows multiple specs to be used with a single set of equations.

The IG900+ database software builds a dependency tree for each specification used in an equation database. This allows the software to recalculate all the equations (and generate tester values) when a spec changes. This is used when specs are updated and when a spec is varied by adjust-until, Specsheet, or Shmoo.

Level Database Example: Level Equations for DRAM Pins

```
(:comment "pin_levels.leveladr")

(use pinset "d1Mx4pins.pindb" d1Mx4_pins)
(range :level (-3v 6v))

(pins :comment "Pin groupings used for levels"
     (default :comment nil :pause nil)
     (pins-form
            (:pin    :power-seq)
            (VCC     1        :pause 1ms)
            (INPUTS  2)
            (DQS     3)
     )
)
(eqset levels_eqns

     (default :vch nil :vcl nil :voh nil :vol nil :vt nil :ioh nil :iol nil)
     (equation-form
            (:pin    :vil     :vih)
            (INPUTS  "Vil"    "Vih")
            (DQS     "Vil"    "Vih"
                     :voh "Voh"
                     :vol "Vol"
                     :ioh "Ioh"
                     :iol "Iol"
                     :vt "(Vih - Vt) > 0.2 ? Vt : (Vih - 0.2)"
                     :vch "Voh + 0.7"
                     :vcl "Vol - 0.7"
            )
     )
     (default)
     (vs-form
            (:pin    :main    :alt                        :ilimit)
            (VCC     "Vcc"    "5.0 - (Vcc - 5.0)"         "300ma")
     )
)
```

Chapter 1.7: Creating Waveform Database Files

Objective: ***Create .waveadr and .wavedb files***
Reference: ***ADR900 Reference Manual***
Programmer's Guide to the Hardware

Introduction

To manipulate the timing (edgeset) hardware based on device specifications, a set of waveform equations is used. These equations calculate test system timing values based on data from the specification databases, set the patgen mode, and set pin modes to be used for a patgen burst. These equations and mode definitions are put into waveform equation databases.

Each pin or pin symbol can have multiple timing values and, typically, one format that can be programmed for each edgeset. In addition, the refresh interval (period) value is programmed by specifying an equation in the waveform database.

The waveform databases can be created using one of four methods:

- ❒ Create an ADR file (.waveadr) containing the ***pins***, ***patgen***, and ***eqset*** statements using a text editor, then translate the ADR file into binary database file (.wavedb) using the ADR900 translator.
- ❒ Use the ADRedit tool to both create and translate the ADR file.
- ❒ Use the Calcsheet tool to create binary database files (.wavedb).
- ❒ Use the Calcsheet tool to create an ADR database file (.waveadr) and manually translate it to create the binary.

Either of the first two methods is recommended for beginning programmers.

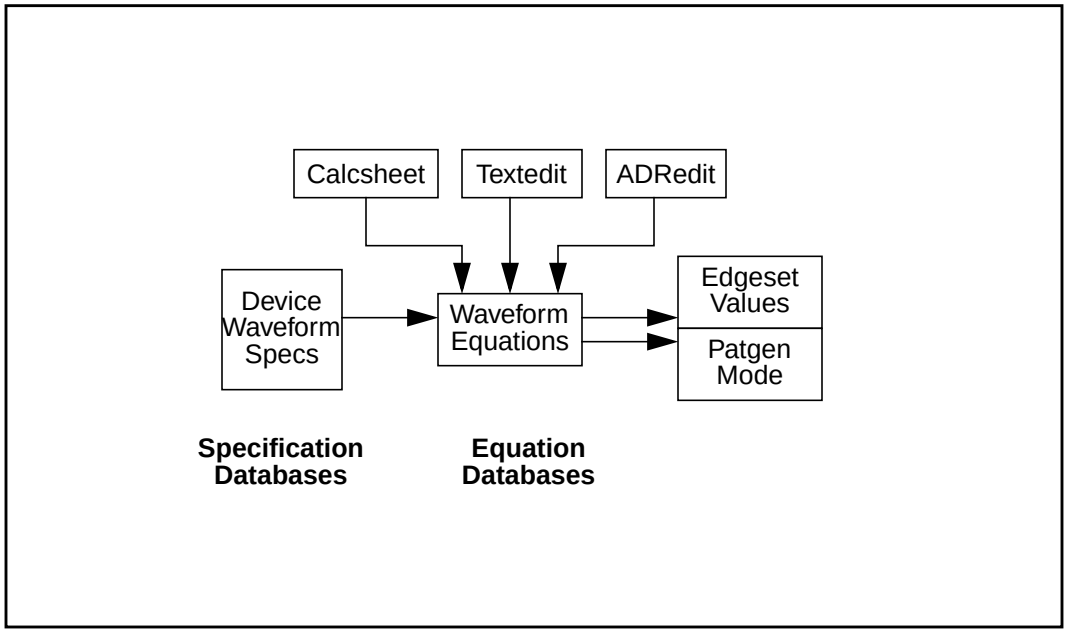


Figure 1.7-2 Relationship of Waveforms Equations to Hardware and Software

Contents of the Waveadr File

The waveform database is defined with the ADR ***pins***, ***patgen*** and ***eqset*** statements, inside a .waveadr file. Only one set of statements is allowed per file. The ***use pinset*** statement must appear before the waveform definitions.

use pinset statement

The ***use pinset*** statement effectively *declares* the pins that can be manipulated by the database file. This declaration must appear before any other statements in the file.

patgen statement

The ***patgen*** statement establishes the patgen frequency mode that this waveform will operate under. This is typically **60MHz** or **100MHz.**

pins statement

The ***pins*** statement defines the modes in which each pin or pin group will be operating. These modes are any of the hardware settings that are available per pin.

For all pin types:

- **chan-mux**: enable/disable channel (pin) muxing
- **mach2**: enable/disable mach2 (turbo) mode
- **driver**: on/off mode for driver prior to patgen burst
- **driver-init**: logic state for driver prior to patgen burst

For drive-only channels only:

- **fmt-mux**: enable/disable for formatter muxing (XYmux)

For IO channels only:

- **zstate**: HiZ (IO switching) mode for the channel
- **scio**: enable/disable same-cycle-io mode
- **comp**: enable/disable for comparator
- **strobe**: window or edge mode for comparator strobe

eqset statement

The ***eqset*** statement contains all the waveform timing equations and formats for each edgeset for each pin on the device. Within the eqset are several clauses:

- **default**: to program non-unique or unused fields
- **dvar-form**: establish default equations for edges
- **tvar-form**: program other timing related hardware that is not included in an edgeset, such as the refresh interval (refresh period).
- **ivar-form**: equations for intermediate variables
- **equation-form**: programming of edge timing and formats per edgeset

Most of the eqset programming is contained in the equation-form. Edgesets are defined here, and will get assigned to an edgeset location in the tester according to the order in which they appear in the database. The first edgeset in the database will be loaded into edgeset 0.

Each pin or pin group can have up to six edges per pin. Drive-only channels have up to six edges per pin when formatter-muxing (d0, d1, d2, d3, d4, d5) and up to three edges when not (d0, d1, d2). IO channels can have up to six edges per pin when doing IO switching (d0, d1, d2, d3, c1, c2), and less when doing just drive or compare within the cycle ((d0, d1, d2) -or- (c1, c2)).

Use pin group symbols whenever possible, as this will allow the software to take advantage of the hardware's parallel loading capability (broadcast write).

Unused edges should normally not be programmed in the edgeset. In some cases unused edges will be automatically placed according to specific guidelines for the system. For example, when using non-return format (NR), the d0 edge will automatically be placed at or after the d1 edge value (when unspecified by the programmer), as this is a specific timing programming guideline for J990 Series Systems. The d0 edge is not used when using NR format, so it does not need to be programmed.

When writing multiple waveform database files, it is important to keep the edgesets loaded at the same edgeset locations in hardware. This means if an specific edgeset will be programmed first (edgeset 0 in hardware), it should appear first in all waveform databases.

The equations for these hardware values can be any valid C expression. These expressions are evaluated (and values calculated) at program load time by the database loader. The specifications used with the equations are determined by the workspace. This allows multiple specs to be used with a single set of equations.

The IG900+ database software builds a dependency tree for each specification used in an equation database. This allows the software to recalculate all the equations (and generate tester values) when a spec changes. This is used when specs are updated and when a spec is varied by adjust-until, Specsheet, or Shmoo.

Waveform Database Example: Wave Equations for DRAM Pins

```
(use pinset "d1Mx4pins.pindb" d1Mx4_pins)
(patgen :freq 60MHz)
(pins :comment "Pin modes for functional tests"
       (default :comment nil :patgen main :mach2 off :chan-mux off
              :driver on :driver-init hi
       )
       (pins-form
              (:pin)
              (ADDRS       :fmt-mux on)
              (RAS         :fmt-mux off)
              (CAS         :fmt-mux off)
              (OE          :fmt-mux off)
              (WE          :fmt-mux off)
              (DQS :io (zstate d0 off d3) :scio off :comp on :strobe edge)
       )
)

(eqset simple_wave_eqns
       (default :comment nil)
       (dvar-form
              (:name                    :equation)
              (default_per  "tRAS + tRP")
              (default_d1   "5.0ns")
              (default_d2   "15.0ns")
              (default_d3   "20.0ns")
              (default_d4   "25.0ns")
              (default_d5   "35.0ns")
       )
; Refresh interval equation
       (tvar-form
              (:name                    :equation    :comment)
              (per_refresh  "tREF"                   "Refresh Interval (Period)")
       )

; Edgeset equations
       (default :comment nil :pin-comment nil)
       (equation-form
              (:edgeset
                           (:pin        :fmt
                                                     (:edge       :equation)
                           )
              )
              (read_edges               :per         "tRAS + tRP"
                           (ADDRS       rc           :fmt1 rc
                                                     (d1          "0")
                                                     (d2          "RAS.d1 + tRAH")
                                                     (d4          "CAS.d1 - tT")
                                                     (d3
                                        "((ADDRS.d4 - ADDRS.d2) / 2 ) + ADDRS.d2")
                                                     (d5          "CAS.d1 + tCAH")
                           )
                           (RAS         pl
                                                     (d1          "ADDRS.d1 + tT")
                                                     (d2          "RAS.d1 + tRAS")
                           )
                           (CAS         pl
                                                     (d1          "RAS.d1 + tRCD_max - tT")
```

```
                                                    (d2          "RAS.d1 + tCSH")
                        )
                        (WE          st
                                                    (d0          "0")
                        )
                        (OE          pl
                                                    (d1          "DQS.c1 - tOE - tT")
                                                    (d2          "DQS.c2 - tOD_min")
                        )
                        (DQS         rc
                                                    (c1          "RAS.d1 + tRAC")
                                                    (c2          "DQS.c1")
                        )
            )

            (write_edges             :per          "@read_edges"
                        (ADDRS       rc            :fmt1 rc
                                                    (d1          "@read_edges")
                                                    (d2          "@read_edges")
                                                    (d4          "@read_edges")
                                                    (d3          "@read_edges")
                                                    (d5          "@read_edges")
                        )
                        (RAS         pl
                                                    (d1          "@read_edges")
                                                    (d2          "@read_edges")
                        )
                        (CAS         pl
                                                    (d1          "@read_edges")
                                                    (d2          "@read_edges")
                        )
                        (WE          pl
                                                    (d1          "CAS.d1 - tT")
                                                    (d2          "WE.d1 + tWP")
                        )
                        (OE          st           (d0          "0")
                        )
                        (DQS         rc
                                                    (d1          "CAS.d1 - tT")
                                                    (d2          "CAS.d1 + tDH")
                                                    (d0          "DQS.d1 - tHIZ - tPW_min")
                                                    (d3          "DQS.d2 + tHIZ + tPW_min")
                        )
            )
      ) ; end equation-form
) ; end eqset
```

Chapter 1.8: Creating Pattern Database Files

Objective: ***Create .svmadr and .svmdb files***
Create .dgenadr and .dgendb files
Create .srcadr and .srcdb files
Create .startadr and .startdb files
Reference: ***ADR900 Reference Manual***
Programmer's Guide to the Hardware

Introduction

For programming flexibility, memory test pattern programming is done in four databases. Each database has a specific purpose, and they can be used in various combinations. Each database type also has a unique file extension.

- *Pattern Control RAM (PCRAM) / Subroutine Vector Memory (SVM) database* — this database is used to program SVM hardware, including pattern controllers, algorithmic memory pattern generators (MPG), and pin state data (vectors).

  File extensions for this database are: .svmadr/.svmdb

- *Data Generator database* — this database is used to program MPG data generator hardware, including data set RAM and data generator selections for device pins.

  File extensions for this database are: .dgenadr/.dgendb

- *Source Select database* — this database is used to program pin-state data source-selection hardware. The source select hardware is used to determine which pin data sources affect the pin's state.

  File extensions for this database are: .srcadr/.srcdb

- *Pattern Start database* — this database is used to program pattern start conditions, including pattern locations (SVM address) and which pattern controllers (main or alt).

  File extensions for this database are: .startadr/.startdb

These pattern databases are created by editing ADR files then translating them to create binary database files. This can be done by using the ADRedit tool to edit and translate the files or using a standard text editor and manually translating the files.

Pattern Objects

These database files will each contain a database **set *(mpgset, dgenset, srcset, startset*)** in addition to various other programming statements and declarations. These database sets will be named and combined with a ***mutset*** in the workspace to create a pattern. These pattern objects are included in the job when it is linked and loaded.

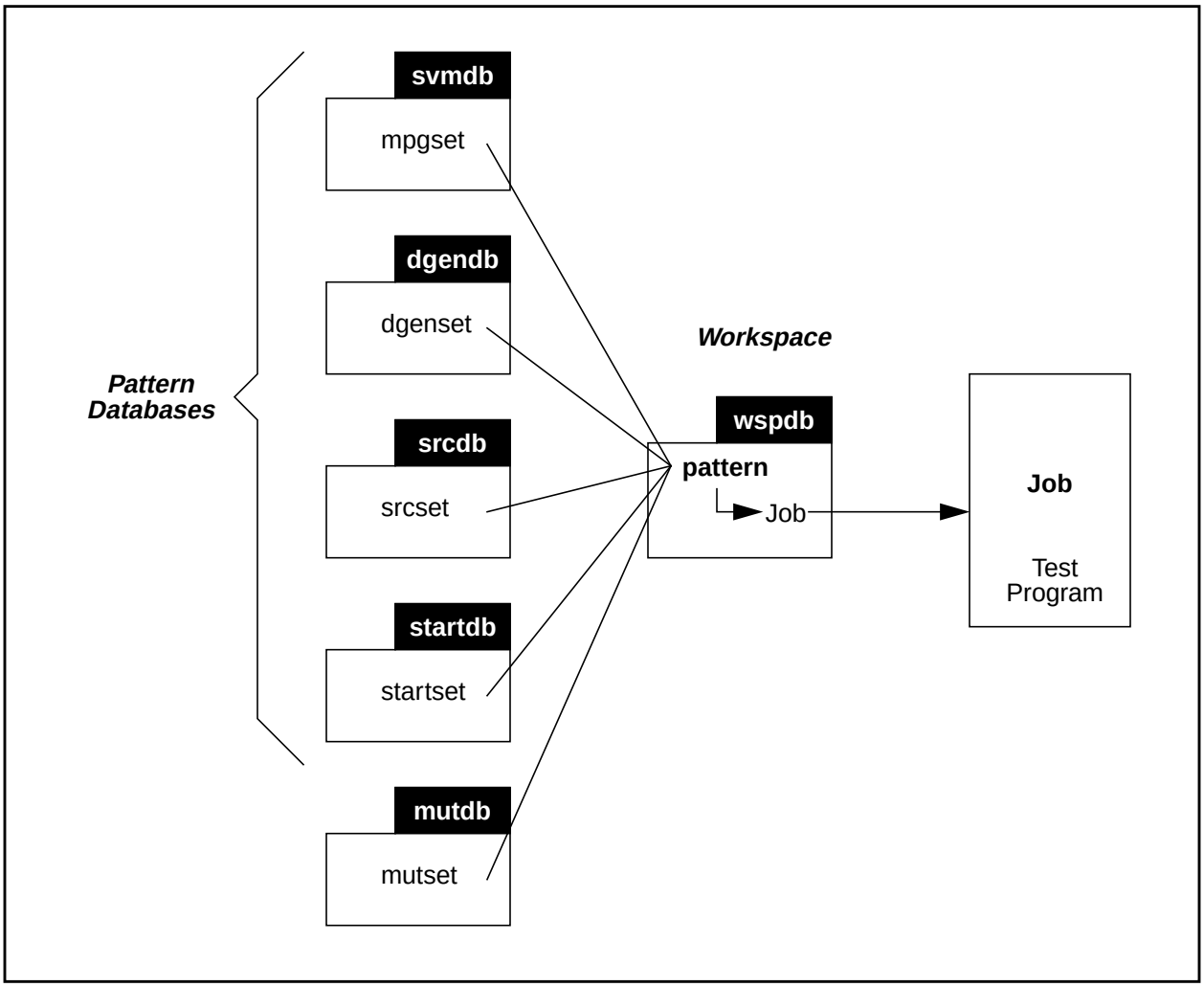


Figure 1.8-1 Relationship of Patterns Databases to Workspace and Job

There are relationships between the various pattern databases that will determine the order in which they should be created. Figure 1.8-2 shows the best sequence for creating the databases.

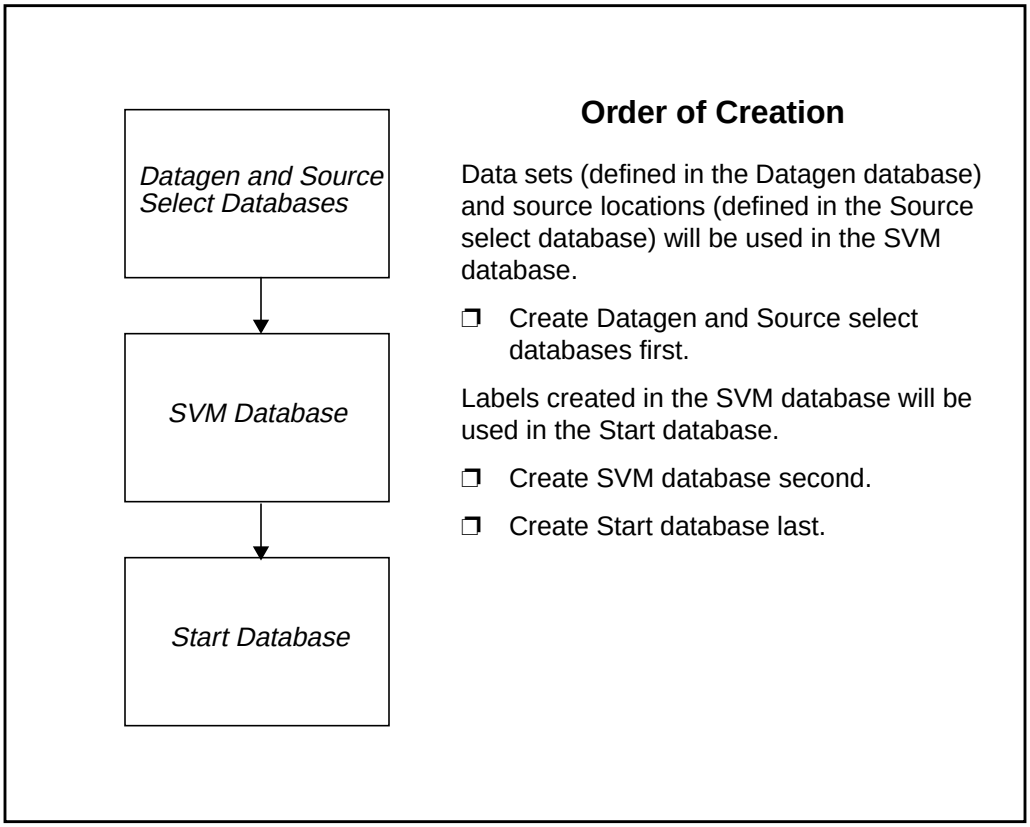

Datagen and Source Select Databases
SVM Database
Start Database
Order of Creation
Data sets (defined in the Datagen database) and source locations (defined in the Source select database) will be used in the SVM database.
❒ Create Datagen and Source select databases first.
Labels created in the SVM database will be used in the Start database.
❒ Create SVM database second.
❒ Create Start database last.

Figure 1.8-2 Order of Creation of the Patterns Databases

Contents of the Dgenadr File

The Data Generator database is defined with the ADR ***dgenset*** statement, inside a ***.dgenadr*** file. Only one ***dgenset*** statement is allowed per file. The ***use pinset*** statement must appear before the data generator programming.

One dgenset contains all the conditions for data set ram, coincidence masking, logic function generator input selections, and pin selections of data generator outputs. These are programmed in separate statements within the dgenset.

map-pin statement

Each pin is assigned to a data generator output, either ***dgen0*** or ***dgen1***.

coin-enablemask statement

Programs the coincidence masking register to a value that is based on the address size of the device. The coincidence masking register is used when data as a function of address coincidence is needed (X=Y). Only those address bits enabled should be considered for coincidence.

lfg-form statement

Programs the input selections to the logic function generators (LFG's). The inputs into the LFGs are patgen address bits x0..x15, y0..y15). For example, a checkerboard data pattern will typically select the LSB for both row and column addresses (x0, y0).

data-form statement

Programs each group of the data set RAM hardware. Data set RAM locations will be used in the SVM database, defining which data set will be used in each pattern cycle. Sets within the data groups will be programmed by nesting parenthesis.

```
(data-form
      (:group
            (:set :dgen0   :dgen1   :lfg0    :lfg1)
      )
      (0
            (0    <...  data set 0 group 0  ...>  )
            (1    <...  data set 1 group 0  ...>  )
      (1
            (0    <...  data set 0 group 1  ...>  )
            (1    <...  data set 1 group 1  ...>  )
      )
      (2
            (0    <...  data set 0 group 2  ...>  )
            (1    <...  data set 1 group 2  ...>  )
      )
)
```

Data Generator Database Example: Pattern Data for 1Mx4 DRAM

```
(use pinset "d1Mx4pins.pindb" d1Mx4_pins)

(dgenset pattern_data
    (map-pin
           :dgen0 (DQ0 DQ2)
           :dgen1 (DQ1 DQ3)
    )

    (coin-enablemask :dgen0 #x3ff :dgen1 #x3ff)

    (lfg-form
           (:lfg    :a       :b)
           (0       x0       y0)
           (1       x0       y0)
    )

    (data-form
           (:group
                  (:set    :dgen0   :dgen1   :lfg0    :lfg1)
           )

           ; solid data: set0=0 set1=1
           (0
                  (0       0        0        "0"      "0")
                  (1       1        1        "0"      "0")
           )

           ; checkerboard data based on lsbs of address
           (1
                  (0       lfg0     lfg1     "a^b"    "~(a^b)")
                  (1       lfg0     lfg1     "~(a^b)" "a^b")
           )

           ; coin data
           (2
                  (0       coin     coin     "0"      "0")
                  (1       ~coin    ~coin    "0"      "0")
           )
    )
)
```

Contents of the Srcadr File

The Source select database is defined with the ADR ***srcset*** statement, inside a ***.srcadr*** file. Only one ***srcset*** statement is allowed per file. The ***use pinset*** statement must appear before the source select programming.

One ***srcset*** programs the conditions for the source select ram for each pin being used. The source select RAM determines what pin data (ADS or SVM) will affect each pin state, which ultimately determines what a pin function is (Address, Clock, or Data).

Within the ***srcset*** statement is a ***src-form*** table. The table has the source selections for each pin. There are three source selections for each pin, and each source selection should be programmed. The selections will depend on the desired pin function.

- ❒ Selection ***vm*** designates the bit to come from vector memory (SVM / PCRAM in the J990 Series).
- ❒ Selection ***ads*** designates the bit to come from the ADS bus.
- ❒ Selection ***0*** designates the bit to be held low.

Selections are programmed in MSB — LSB order. Recommended source selections per pin funCtion and mode are contained in the table below.

Table1.8.1
Recommended Source Selections: Drive-Only Pins

Description	bit2	bit1	bit0
Address (from memory pattern generator)	0	0	ads
Address (from SVM pin data)	0	0	vm
Data input (from data generator)	0	0	ads
Clock (normal)	0	0	vm
Clock (mach2)	0	vm	vm

Table1.8.2
Recommended Source Selections: IO Pins

Description	bit2	bit1	bit0
Address (from memory pattern generator)	vm	vm	ads
Address (from SVM pin data)	vm	vm	ads
Data input (from data generator)	vm	vm	vm
Clock (normal)	vm	vm	vm
Clock (mach2)	vm	vm	vm
SCIO Address (from memory pattern generator)	vm	ads	ads
SCIO data input (from data generator)	vm	ads	ads

When using drive-only pins in the formatter-mux mode, the database software programs the *X* and the *Y* half of the channel to have the same selections.

Source select RAM location 0 is reserved by Teradyne for programming all bits to come from vector memory (vm). User programmed labels are loaded starting at location 1.

Source Select Database Examples

Pin Data Selections for DRAM Pins

```
(use pinset "d1Mx4pins.pindb" d1Mx4_pins)

(srcset dram_srcset
      (src-form
            (:label     ADDRS       DQS           CLKS)
            (srcloc1    (0 0 ads)   (vm vm ads)   (0 0 vm))
      )      ;for mach2 operations CLKS should be (0 vm vm)
)
```

Multi-Function Pin: Address and Clock

```
(use pinset "mfpins.pindb" multi_function_pinset)

(srcset mfpin_srcset
      (src-form
            (:label     AC_PIN      DQS           CLKS)
            (addr_func  (0 0 ads)   (vm vm ads)   (0 0 vm))
            (clk_func   (0 0 vm)    (vm vm ads)   (0 0 vm)))
)           ;for mach2 operations CLKS should be (0 vm vm)
```

Contents of the Svmadr File

The SVM (PCRAM) database is defined with the ADR ***mpgset*** statement, inside a ***.svmadr*** file. Only one ***mpgset*** statement is allowed per file. The ***use pinset*** statement must appear before the SVM definitions.

The SVM database will also contain an ***export*** statement, to allow other modules and databases to reference SVM labels that it defines. It will also typically contain an ***import*** statement, so it can reference ***edgeset*** and ***srcset*** labels within the patterns.

The ***mpgset*** contains a ***vector-form*** statements, which contains the ***default*** and actual pattern information for each location of SVM. A header is used to define the table programming layout for the ***vector-form***. Pattern ***labels*** are defined by inserting pattern names into the vector-form. Multiple patterns (labels) are usually defined in one mpgset.

```
(mpgset
      ( ... cycle definition header ...)
      (vector-form
            (default ...)
      LABEL
            ( .. pattern step for one SVM location...)
            ( .. pattern step for one SVM location...)
            ( .. pattern step for one SVM location...)
            ( .. pattern step for one SVM location...)
      )
)
```

mpgset cycle definition header

The header is used to declare the fields that will be programmed in tabular format (and their order). It includes a list of pins with vector programming suffixes, which establishes the vector states that will be allowed for each pin in each cycle.

```
    (:e   ADDRS.M  CLKS.S   DQS.S :mpg  :op)  ;header
(vector-form
    (E    X        X        X     (...) (...)) ;cycle
    (E    X        X        X     (...) (...)) ;cycle
    (E    X        X        X     (...) (...)) ;cycle
    (E    X        X        X     (...) (...)) ;cycle

)
; where:
;   E is the edgeset for this location
;   X is the cycle's vector state for a pin symbol
```

Use pin group symbols whenever possible, as this will allow the software to take advantage of the hardware's parallel loading capability (broadcast write).

default

Similar to other ADR programming statements, the default clause defines the default conditions for each step in the vector-form statement. It is recommended that you have a default for each SVM and MPG field.

Pattern Locations

Each pattern location contains pin state information (vectors), and memory patgen (MPG) operations, and SVM opcodes (flow control). These are grouped by parentheses.

```
label
      ; one pattern step
      ( ... pin state information / vectors ...
            (... Memory Patgen operations ...)
            (... SVM flow control ...)
      )
```

A label is a way to create a name for a pattern RAM location.

Basic Vector Programming

The pin state data for each cycle is referred to as a vector. Depending on the pin's mode and channel type, various vector states are available. These vectors are stored in the per-pin SVM. Each of the available states is programmed with a symbol.

IO channels have three bits of vector data which drive an eight state model. For example, when in the normal mode, the eight available IO channel states are:

Table1.8.3
Eight State Model

State	Symbol	Description
0	0	Drive 0 data
1	1	Drive 1 data
2	M	Expect midband voltage (between Voh and Vol)
3	V	Expect valid voltage (above Voh or below Vol)
4	L	Expect logic low
5	H	Expect logic high
6	W	Hold window strobe
7	X	Don't care

IO channel vectors are most commonly programmed in this symbolic mode, where the state symbol is either **0**, **1**, **M**, **V**, **L**, **W**, or **X**, and the pin symbol has the **.S** extension in the vector- form header. The single character vector after the period (.) will cause the vector to apply to all pins in the pin symbol.

```
(vector-form
      (IOS.S)          ; header using symbolic vector
      ( .0 )           ; drive data in this cycle
      ( .L )           ; expect data in this cycle
)
```

When using algorithmic address or data to drive the channel, the states of the channel are driven partially by the ADS bus. The lsb of the three bit input to the 8 state model will come from the ADS bus. The bit on the ADS bus will then determine whether the channel expects a 0 or a 1, or drives a 0 or a 1. States 0,1, L, and H in the SVM will be most commonly used for this application. Either state 0 or 1 can be used when driving data that is coming from the ADS bus, either state L or H can be used when expecting data from the ADS bus.

Drive-only channels can have four bits of data that drive the channel state. However, the number of bits that are used depends on the mode of the channel.:

Table1.8.4
Drive Only Channels — Vector Bits

Fmt-Mux	Mach2	Number of Vector Bits Used (which ones)
Off	Off	1 (bit 0)
On	Off	2 (bits 0 and 1)
Off	On	2 (bits 0 and 2)
On	On	4 (bits 0, 1, 2, and 3)

When doing formatter-muxing and Mach2, two bits are used for the first half of the cycle, and two bits are used for the second half of the cycle. When not doing formatter- muxing, only the first two bits can be used. When the channel is not in mach2 mode, only half of the available bits are used.

Drive-only channel vectors are programmed in the memory mode, where the state symbol is either **0**, **1**, **2**, or **3**, and the pin symbol has the **.M** extension in the vector-form header. The single character vector after the period (.) will cause the vector to apply to all pins in the pin symbol.

```
(vector-form            ; no muxing or mach2 (clock pin)
      (WE.M)            ; header using memory vector
      ( .0 )            ; drive 0 in this cycle
      ( .1 )            ; drive 1 in this cycle
)
```

When using algorithmic address or data to drive the channel, the drive data is supplied by the ADS bus. In this case, the memory vector should be 0.

Vector Programming with Macros

Since memory test patterns often use repetitive vector states, these are easily programmed with macros. Use the C preprocessor ***#define*** statement to define the header and the states for each vector for each cycle type. This effectively defines the cycle definitions that will be used in the pattern.

```
; cycle definition macros

#define CYCLE_DEFINITION \
          :e               :ign    DQS.S   RAS.M    CAS.M   WE.M    OE.M

#define RAS_ONLY_CYCLE\
          ras_only_edges   1       .X      0        1       1       1

#define READ_CYCLE\
          read_edges       0       .L      0        0       1       0

#define WRITE_CYCLE\
          write_edges      1       .0      0        0       0       1

#define INACTIVE_CYCLE\
          inactive_edges   1       .X      1        1       1       1


(mpgset dram_patterns

    (vector-form (CYCLE_DEFINITION :mpg :op)

            (default INACTIVE_CYCLE
                     ...........
            )
    pattern_name_label
            ( WRITE_CYCLE ( ..mpg operations.. ) ( ..flow control opcodes.. ) )
            ( READ_CYCLE  ( ..mpg operations.. ) ( ..flow control opcodes.. ) )
            ( INACTIVE_CYCLE ( ..mpg operations.. ) ( stop ) )
    ) ; end vector-form
) ; end mpgset
```

SVM Database Example: 1Mx4 DRAM March Pattern

```
(use pinset "d1Mx4pins.pindb" d1Mx4_pins)

(export :label ( March ) )

(import :edge-label (
     ras_only_edges
     read_edges
     write_edges
     inactive_edges
     )
)

(import :src-label srcloc1)

#define XMAX                  1023
#define YMAX                  1023
#define FULL_ADDR_SIZE        1048576
#define RPT_FULL_ADDR_SIZE    1048575
#define RPT_WAKEUPS           7

; cycle definition macros

#define CYCLE_DEFINITION \
          :e              :ign    DQS.S   RAS.M    CAS.M   WE.M    OE.M

#define RAS_ONLY_CYCLE\
          ras_only_edges  1       .X      0        1       1       1

#define READ_CYCLE\
          read_edges      0       .L      0        0       1       0

#define WRITE_CYCLE\
          write_edges     1       .0      0        0       0       1

#define INACTIVE_CYCLE\
          inactive_edges  1       .X      1        1       1       1

; Notes:
; - ADDRS are not mentioned here, since data actually comes from ADS bus due to
;         source select programming.  No pin state data is required.
; - DQS are always programmed to drive 0 (.0) when writing, and expect 0 (.L)
;   when reading, but actually data comes from ADS bus due to source select
; - all clocks (RAS, CAS, WE, OE) are always programmed to drive the logic state
;   that is needed for a particular cycle type.  For starting a page mode
;   operation, the logic state for RAS is 0, and for ending a page the RAS
;   state is 1.
```

SVM Database Example (continued): DRAM March Pattern

```
(mpgset dram_patterns

     (vector-form (CYCLE_DEFINITION :mpg :op)

     (default INACTIVE_CYCLE
             :src srcloc1 :stv 0
             (
             :xreg 0 :yreg 0
             :xa hold :xb hold :xc hold :xd hold
             :ya hold :yb hold :yc hold :yd hold
             :xconst-next hold :yconst-next hold
             :xena-next hold :yena-next hold
             :xaluj zero :xaluk zero :xaluop xor
             :yaluj zero :yaluk zero :yaluop xor
             :xdevadr xa :ydevadr ya :xdevinv off :ydevinv off
             :xdgadr xa :ydgadr ya :xdginv off :ydginv off
             :dgroup-next hold :dset 0
             :disp-push-failcntr off :disp-ign-fails off :disp-clr off
             :catch-clr off
             :mutsim idle :mutsim-inv off
             )
             (
             refresh-disable
             )
     )
; March pattern
;    - solid data (dgroup 0)
; 1) preset a counters to max address for use in a forward march,
;     while performing wakeup cycles
; 2) fast x scan write data set 0
; 3) fast x incr read data set 0
; 4) write data set 1
; 5) fast x scan read data set 1 max to min.

March
     (RAS_ONLY_CYCLE
             (
             :xreg XMAX :yreg YMAX
             :xa load-reg :ya load-reg
             :xena-next jam-reg :yena-next jam-reg
             :dgroup-next 0
             )
             (rep RPT_WAKEUPS)
     )
```

SVM Database Example (continued): DRAM March Pattern

```
    (WRITE_CYCLE
            (
            :xa inc :ya inc-link
            :xdevadr xa :ydevadr ya
            :xdgadr xa :ydgadr ya
            :dset 0
            )
            (rep RPT_FULL_ADDR_SIZE)
    )

    (READ_CYCLE
            (
            :xa inc :ya inc-link
            :xdevadr xa :ydevadr ya
            :xdgadr xa :ydgadr ya
            :dset 0
            )
            (loop FULL_ADDR_SIZE)
    )

    (WRITE_CYCLE
            (
            :xa hold :ya hold
            :xdevadr xa :ydevadr ya
            :xdgadr xa :ydgadr ya
            :dset 1
            )
            (eloop .-1)
    )

    (READ_CYCLE
            (
            :xa inc :ya inc-link
            :xdevadr xa :ydevadr ya
            :xdgadr xa :ydgadr ya
            :dset 1
            )
            (rep RPT_FULL_ADDR_SIZE)
    )

    (INACTIVE_CYCLE ()
            (stop)
    )

    ) ; end vector-form
) ; end mpgset-form
```

Using the eloop Opcode

This section provides additional information on the use of the eloop (end loop) opcode in svmset statements. In particular, some common mistakes in the use of eloop produce runtime errors; this section describes how these segments of code can be written to avoid these errors.

The eloop Opcode

The syntax for the eloop opcode is:

```
eloop [VLABEL]
```

This opcode, which must be paired with a loop opcode, will decrement the loop counter and branch to VLABEL if the counter is non-zero. This may be a branch to a vector with a loop opcode. Note, however, that eloop blocks any pushes on the loop stack for the next cycle. If the branch is to a loop N opcode, a nop is performed instead.

If VLABEL is omitted, the branch address is set to the previous unmatched loop N address.

When the loop counter is zero, a "pop loop stack into the loop counter and continue to the next vector" occurs.

Typical Usage

Typically, the eloop opcode will be matched with an opening loop N opcode. The following is an example of this usage.

```
loop 512 ; loop start location
      .
      .     ; other code
      .
eloop ; loop end location
```

In this example the loop will be repeated 512 times.

It may be desirable to eloop to a different location than the opening loop N opcode. An example of this is shown below.

```
            loop 512
            .
            .
            .
LABELX      .
            .
            .
            eloop LABELX
```

Now the code between the two *LABELX* steps will be looped on 512 times. The code between the *loop 512* step and the first *LABELX* step will be executed once the first time through the loop. If this action is desired, take care that you do not eloop to a *rep* opcode or a *nested* loop N opcode.

Common Mistakes

Here is an example which will cause a stack underflow error:

```
(                          (loop YMAX))
       .
       .
       .
(read        :xa inc      (rep XMAX ))
(read        :ya inc      (eloop .-1))
```

XMAX in this example will not be pushed on the stack because the eloop blocks the push attempted by the rep statement. A stack underflow error will occur.

Another erroneous example is nested loops as shown below:

```
              (
                     (
                     :xreg XMIN
                     :yreg YMIN
                     :xa jam-reg
                     :ya jam-reg
                     )
                     (loop HEIGHT)
              )
XLOOP         (write 0            (loop WIDTH))
              (read 0             (if (flags) goto OUT_OF_HERE))
              (write 1 :ya inc    (eloop XLOOP))
              (inactive
                     (
                     :xa inc
                     :yreg YMIN
                     :ya jam-reg
                     )
                     (eloop XLOOP)
              )
OUT_OF_HERE          (loop_pop)
                     (loop_pop)
```

In this example, WIDTH will not be pushed on the stack, and a stack underflow error will occur.

Corrected Examples

An approach that you can take if you want to loop back to a rep step is to add a SVM step before the rep step and loop to the earlier step. An example of this is shown below.

```
(                          (loop YMAX))
       .
       .
       .
(dummy                     ()           )     ; extra SVM step here
(read        :xa inc       (rep  XMAX))       ; now repeat to XMAX
(read        :ya inc       (eloop .-2))
```

A variant of this approach is shown next (notice the change in the number of repeat steps).

```
(                       (loop YMAX)   )
       .
       .
       .
(read        :xa inc    ()            )     ; extra SVM step here
(read        :xa inc    (rep XMAX-1)  )     ; repeat to XMAX - 1
(read        :ya inc    (eloop .-2)   )
```

For nested loops, the best approach is to split the eloops so that each loop references a different VLABEL. One approach using a dummy step is shown below:

```
             (                         (loop YMAX)  )
XLOOP2       (dummy                    ()           ) ; extra SVM step
XLOOP        (            :xa inc      (loop XMAX)  ) ; loop on XMAX
                    .
                    .
                    .
             (                         (eloop XLOOP))
             (            :ya inc      (eloop XLOOP2))
```

A variant of this approach that does not use a dummy step is shown next. The WIDTH must be reduced by one.

```
             (
                    (
                    :xreg XMIN
                    :yreg YMIN
                    :xa jam-reg
                    :ya jam-reg
                    )
                    (loop HEIGHT)
             )
XLOOP 1      (write 0                  )
             (read 0                   (if (flags) goto OUT_OF_HERE_1))
             (write 1 :ya inc)
XLOOP2       (write 0                  (loop WIDTH -1))
             (read 0                   (if (flags) goto OUT_OF_HERE_2))
             (write 1 :ya inc          (eloop XLOOP2))
             (inactive
                    (
                    :xa inc
                    :yreg YMIN
                    :ya jam-reg
                    )
                    (eloop XLOOP1)
             )
OUT_OF_HERE_2              (loop_pop)
OUT_OF_HERE_1              (loop_pop)
```

Pattern Scaling

Pattern scaling allows patterns (located in svmadr files) to contain expressions which can be changed at runtime. The pattern can modify primary operands (loop counts, repeat counts, utility counters) or MPG initializers (:xreg, :yreg), allowing a number of different size patterns to be derived from one svmadr file.

Step 1. Create runtime expressions

These runtime expressions are entered as a string of characters enclosed in braces { }. They can contain any legal C++ expression of variables and arithmetic symbols. The resulting integer value is used in a writesys(mpg_op,) call to update hardware.

Within a pattern object the following fields can accept runtime expressions:

```
Pattern Control Opcodes
      set cN {expression}
      loop {expression}
      rep {expression}
MPG Instructions
      :xreg {expression}
      :yreg {expression}
```

Step 2. Import the variables used in the runtime expressions

The variables must be imported into the svmadr file. The following are examples of the syntax:

```
(import  :param-int a1)
         or
(import  :param-int (a1 a2 a3))
```

Variables defined by the import statements do not have to be declared anywhere else in the program. The compiler will declare them. The string that creates the runtime expression is not retained in the svmdb file. To edit a runtime expression you must modify the svmadr file not the svmdb file. (Note that this modification method is not the same as expressions used in waveadr or leveladr equations. Those equations can be modified at runtime in the db file.)

Step 3. Run adr900

The translator and compiler creates a function for each pattern when the svmadr file is translated. First the translator will create the function <mpgset>_update.t where <mpgset> is the name of an mpgset. This function is passed the arguments that set the expressions. The translator also creates the file <mpgset>_update.h to declare the function. After translating, the at900 compiler is automatically invoked to compile the <mpgset>_update.t, creating a <mpgset>_update.o file. The job's makefile does not have to be altered for this. All three <mpgset>_update files will be created in whatever directory the svmdb file is created in.

Step 4. Add <mpgset>_update.o file to the workspace database

The function created (i.e. the <mpgset>_update.o file) must be included in the job's workspace. This is shown in the example provided on page 1.8.21

Step 5. Create a scaling function to call the compiler-generated function

This function must be executed before the pattern which is to be scaled is run. It can be executed by including it in template code, in your main.t file, or elsewhere. If your mpgsets are combined into an svmtable, be sure to call all the scaling functions for all mpgsets.

The function generated by the compiler will have the following arguments:

```
void <mpgset>_update (
      TT_VEC_SVM_UPLOAD tt_upread,
      TT_VEC_SVM_UPLOAD tt_upwrite,
      TT_CM_VAL tt_patgen,
      int a1,
      int a2,
      int a3,
         .
         .
         .
      int aN)
```

The tt_upread argument is the same as `readsys(mpg_op,...)` namely:

```
TT_VEC_SVM_DOWNLOAD_IMAGE     ;SVM copy only
TT_VEC_SVM_ONLY               ;SVM memory only
```

Set this to the desired source of the data.

The tt_upwrite argument indicates the recipient of the altered data. It is like tt_upread, with the addition of a modify both option:

```
TT_VEC_SVM_DOWNLOAD_IMAGE;    ;svm copy only
TT_VEC_SVM_ONLY               ;SVM memory only
TT_VEC_SVM_MODIFY_BOTH        ;Both SVM and copy
```

The tt_patgen argument is one of: (see writesys(mpg_op,...))

```
TT_CM_PATGEN0     ;main patgen
TT_CM_PATGEN1     ;alt patgen
```

The param-int arguments are copied in the order they appear as arguments 4 to n+4, where n is the number of param-int arguments.

The scaling function call will then look something like this, where v1 to vN are values to bind to a1..aN

```
<mpgset>_update (TT_VEC_SVM_DOWNLOAD_IMAGE,
      TT_VEC_SVM_DOWNLOAD_IMAGE, TT_CM_PATGEN0, v1,
      v2, v3, ..., vN);
```

There are several ways to alter the pattern, set by the first argument to this function. If the download image is changed (TT_VEC_SVM_DOWNLOAD_IMAGE), the function can be executed at any time. If the SVM image is changed (TT_VEC_SVM_ONLY), it should be done after the pattern is loaded (In an on_patgen_start function or after a writesys(select_pattern)).

Example

test.svmadr

```
(import     :param-int  (
            xmin
            xmax
            ymin
            ymax
)
(mpgset     Scan
            (vector-form
                  (:xreg {xmin} :yreg {ymin}
                  )    (loop {(xmax-xmin)*(ymax-ymin) + 1)})
                  ()   (rep {(xmax-xmin)*(ymax-ymin)/2 - 1}
            )
)
```

workspace.wspadr

```
(job Four_Meg_DRAM (
      Scan_update.o
      )
)
```

func_Template.t

```
#include "Scan_update.h"

      xmin = 1023;
      xmax = 1023;
      ymin = 512;
      ymax = 512;
      Scan_update (TT_VEC_SVM_DOWNLOAD_IMAGE,
            TT_VEC_SVM_DOWNLOAD_IMAGE, TT_CM_PATGEN0, xmin,
            xmax, ymin, ymax);
```

Contents of the Startadr File

The pattern start database is defined with the ADR ***startset*** statement, inside a ***.startadr*** file. Only one ***startset*** statement is allowed per file.

A startset object is actually defined by a the ***:name*** clause within the startset statement, so in effect, one startset statement can create multiple startset objects. The names of these startset objects will be placed into the workspace.

A startset includes where the pattern will start (label), for both main and alt pattern controllers. Not all pattern labels need to be put into a startset, only those labels which will be starting locations for patterns.

Currently the startset requires programming of fields not used by the J990 Series hardware. These fields (relating to LVM hardware on the J971 system) should be programmed to ***nil.***

Pattern Start Database Example

```
(:comment "patterns.startadr")

; create startsets march, scan, addrcompl, fast_page,
;        refresh_loop, read_write_loop, and page_loop

(startset
         (default  :start-alt nil  :svm-alt nil )
         (start-form
                  (:name            :svm)
                  (march            March)
                  (scan             Scan)
                  (addrcompl        Addrcompl)
                  (fast_page        Fast_page)
                  (read_write_loop  Read_write_loop)
                  (refresh_loop     Refresh_loop)
                  (page_loop        Page_loop)
         )
)
```

Adding Patterns to a Test Program

When patterns are to be added to a test program, usually only the SVM and Start databases need to be changed. The data generator and source select databases are usually unchanged. The typical sequence is to add the pattern to an existing ***mpgset***, then create a pattern ***startset*** in the Start database.

Once the pattern has been created, it is added to the job workspace.

Chapter 1.9: Creating Calibration Database Files

Objective: ***Create .caladr and .caldb files***
Reference: ***ADR900 Reference Manual***
AEL Reference Manual

Introduction

Calibration for J990 series test systems is controlled by system software called AEL™ (Automatic Edge Lock). AEL uses TDR techniques to program ramp slopes, DC correction values, and deskew DACs to compensate for variations in circuit components and channel path propagation.

Calsets allows AEL to calibrate some system resources to the specific requirements of the test program. Every test program needs at least one calset to guide AEL to its optimum calibration.

Calsets calibrates the following parameters:

- ❒ timing deskew for d0, d1, d2, d3, d4, d5
- ❒ strobe modes (edge and window) for c1 and c2.
- ❒ multiplexing odd and even channels
- ❒ multiplexing within the formatters
- ❒ DC levels (VIH, VIL, etc.)

Up to fifteen calsets can be programmed for a given test program; however, only one calset is in effect at any given time. The other calibration values will be stored until needed. Note however, that each calset adds to total calibration time. Therefore, do not put unnecessary calsets in a test program.

Creating Calsets

To create a calset and add it to your test program, do the following:

1. Create a *.caladr containing one or more calsets. ADRedit can supply the complete framework for the file, and can also be used to create the file. The file can also be created or edited with any text editor.

2. Translate the caladr file to a caldb file using the **adr900** command. For example:

   > **adr900 dram_calsets.caladr**

   The file produced by the ADR translator will be given a caldb extension; however, it will still be an ASCII file, not a binary file. The caldb file will be organized in the precise format used by the AEL program. Do not edit the caldb file. If later changes are required, modify the **caladr** file and retranslate.

3. Include the caldb file in your workspace. The caldb file name is placed in the job group in the .wspadr file. For example,

```
(job            j993_1Mx4
                        (
                        explicit_timing         ; waveform
                        wave_timing_70ns        ; waveform
                        wave_timing_80ns        ; waveform
                                .
                                .
                                .
                        "d1Mx4_calsets.caldb"   ; calsets
                                .
                                .
                                .
                        )
)
```

   Note that the file name is enclosed in quotes.

4. Add appropriate calsets to your job flow by including them in instances using the Insted tool. For example

```
(define-instance icc_standby_low_power ivs_Template
        (value levels icc_low_power_levels)
                        .
                        .
                        .
        (value calset icc_level_cal)
.
                        .
                        .
                        .
)
```

   After the job program is loaded and run is entered for the first time, the caldb file will be copied to an AEL configuration file, and AEL will be executed.

Caladr Example: 1Mx4 DRAM

```
(use pinset "d1Mx4_pins.pindb" d1Mx4_pins)

(range :level (-2v 7v))

(calset nominal_level_cal
          (patgen                 :freq 60Mhz)
          (default
                                  :vil 0.0
                                  :vih 3.0
                                  :chan-mux off
                                  :fmt-mux off
                                  :optimize d3_drive
                                  :strobe edge
          )
          (cal-form     (:pin             :vil   :vih   :fmt-mux)
                        (CLKS             0.0    3.0    off)
                        (ADDRS            0.0    3.0    on)
          )
          (cal-form     (:pin             :vil   :vih   :strobe   :optimize)
                         (DQS             0.0    3.0    edge      (d0_drive d3_drive))
          )
)

(calset spec_level_cal
          (patgen                 :freq 60Mhz)
          (default
                                  :vil 0.0
                                  :vih 3.0
                                  :chan-mux off
                                  :fmt-mux off
                                  :optimize d3_drive
                                  :strobe edge
          )
          (cal-form     (:pin             :vil   :vih   :fmt-mux)
                        (CLKS             0.8    2.4    off)
                        (ADDRS            0.8    2.4    on)
          )
          (cal-form     (:pin             :vil   :vih   :strobe   :optimize)
                         (DQS             0.8    2.4    edge      (d0_drive d3_drive))
          )
)
```

Notes

This example is a caladr file containing two calsets. For this device, format multiplexing will be done for the device's address pins. This will allow two independent address data bits (X and Y) to be generated in the same cycle by each address channel. This is a normal requirement for address-multiplexed DRAMs. This mode is used exclusively throughout the job execution. With all pins remaining in the same mode, the calibration mode does not change.

```
(use pinset "d1Mx4_pins.pindb" d1Mx4_pins)
(range :level (-2v 7v))
```

The pinset specifies which test head resources need to be calibrated for the test program. Range indicates which DC levels range will be used by the test program. This is a choice among three 9-volt ranges: -4V to 5V, -3V to 6V, and -2V to 7V.

```
(calset nominal_level_cal
          (patgen                 :freq 60Mhz)
```

The first calset, the nominal_level_cal, contains nominal cal levels. In the demo DRAM program, this calset is used with the basic functional and nominal function instances. Other calsets are used with tight timing instances such as those for characterizing specific parameters (such as t_{RAC}).

The patgen clause tells AEL at what frequency the patgen will be running. This is a choice of 60Mhz, 100Mhz, or dual_60Mhz. This choice is test system dependent, and should match what isprogrammed in the waveadr file.

```
          (default
                                  :vil 0.0
                                  :vih 3.0
                                  :chan-mux off
                                  :fmt-mux off
                                  :optimize d3_drive
                                  :strobe edge
          )
```

Place all items you wish to default in the default clause. Defaulted item can be overridden by specifying a header in the cal-form clause and providing the desired values.

The :vil and :vih keywords calibrate the DC levels. This is typically defaulted to the standard levels at which you will want to set VIL and VIH for your device. The chan-mux and fmt-mux selection can be either off or on.

The optimize selection is used to specify the edge condition for which calibration should be optimized. This is a choice among: d0_patgen_freq, d0_drive, d3_drive, and d3_mux_surround. These are selected based on the application. In this example d3_drive is optimized. This means that AEL would optimize for the d3 edge on an IO channel. The d3 edge is used to change the IO channel from drive (for writing data to the DUT) to HiZ (for reading data from the DUT).

Strobe can be window or edge. Here edge is selected.

Three headers can be omitted from the defaults or the cal-form clause in a caladr. These are :chan-mux, :optimize, and :strobe. If these headers are omitted, AEL will not optimize these selections to the specifics of the test program.

```
        (cal-form     (:pin            :vil   :vih   :fmt-mux)
                      (CLKS            0.0    3.0    off)
                      (ADDRS           0.0    3.0    on)
        )
        (cal-form     (:pin            :vil   :vih   :strobe   :optimize)
                       (DQS            0.0    3.0    edge      (d0_drive d3_drive))
        )
```

The cal-form clause completes the calset. DQs, clocks, and address pins are all included in the :pins list in the cal-form clause. The VIL and VIH values, even though defaulted earlier, are specified again in the cal-form. The new specifications override the default. The VIL and VIH values are the same as the defaults in the first calset, this is to be expected for the nominal_level_cal calset. For a case where the VIH and VIL values are different, see the spec_level_cal calset. For clocks, format multiplexing is not used; however, address channels will use format multiplexing. Therefore, the :fmt-mux header is added to allow address channels to be calibrated in that mode.

For the IO pins, d0 will also be optimized. Notice the parenthesis around the *do_drive d3_drive* syntax. This is a standard ADR technique for adding a list of items under the same header.

Different calsets can be used at various points in the test program flow to adjust calibration values to the optimum for particular tasks.

Chapter 1.10: Creating Workspace Database Files

Objective: ***Create .wspadr and .wspdb files***
Reference: ***ADR900 Reference Manual***

Introduction

Once all the test-condition databases have been created (waveform equations, patterns, levels equations, specifications, etc.), they need to be grouped together to form a test program. This grouping is performed by the workspace database.

The workspace lists all of the databases (sets) and the files (modules) which contain them. This is needed for loading of databases into the test program. When a database is specified, it is loaded from the correct module.

The workspace database also creates the higher level database objects that consist of lower level database objects. For example, combining a specific specset and eqset to create a waveform object. The test program itself is a higher level databases object, which contains all the databases and modules needed for linking a test program. This means that the job is really defined in a workspace.

When the job is loaded, the loader use the workspace to link all of the databases and modules to create the test program. The workspace database name and the job name are both supplied to the loader.

Modular programming allows databases and modules to be shared between test programs. The workspace can contain multiple jobs, which may be appropriate for a family of devices which have similar characteristics and test conditions.

In essence, the Workspace database defines the linkage between the test program and all the databases and executable modules that make it. These linkages are represented by the lines in Figure 1.10-1.

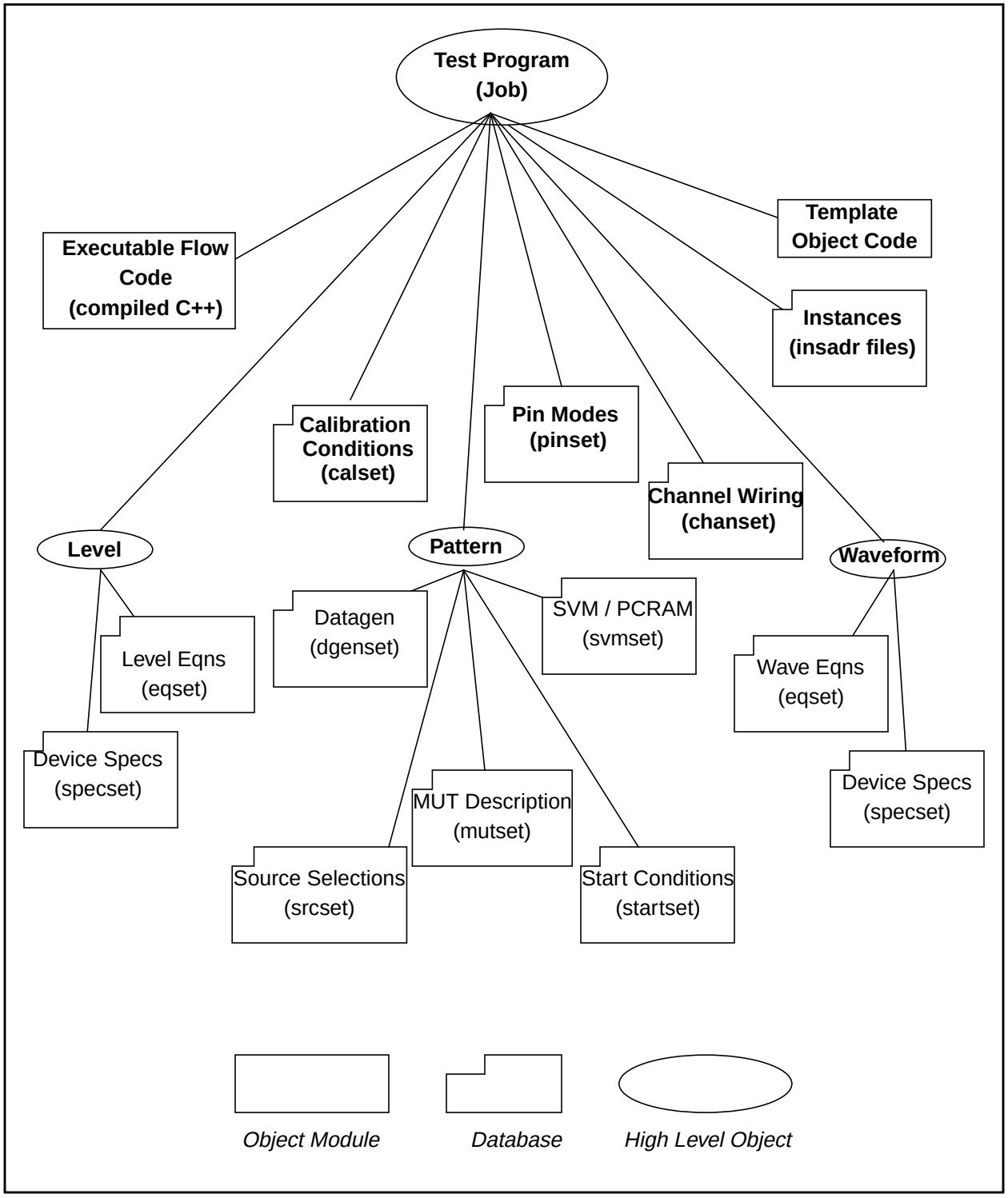


Figure 1.10-3: Conceptual Diagram - Purpose of Workspace Database

Creating the Workspace Database

The workspace database is created by editing an ascii database (***.wspadr***) file, including all the required statements. This can be done with the ADRedit tool or a standard text editor such as ***vi***. The ADR file is then translated to create the binary database file (***.wspdb***).

A script (***build_workspace***) is also supplied with the system software to help build the workspace database file. This script is helpful when creating a workspace from scratch, as it will put in names for all the databases and modules in a directory. It then performs some default grouping of objects to create the job. Some editing of the objects would then be performed by the programmer, to create a fully specified workspace.

An alternative method of creating a workspace is to translate all database files, specifying a workspace database file. During translation, the names of all database sets will be placed into a binary workspace database. That binary workspace database can be converted to ASCII (ADR) form, where it can be edited to create the database groupings.

Contents of the Workspace Database

The Workspace database consists of ***set*** declarations (*chanset*, *pinset*, etc.) and ***group*** definitions (*waveform*, *level*, *job*, etc.) inside a ***.wspadr*** file. The test program (job) is defined as a group object, and multiple job groups can be defined in one workspace. The job group object contains all the databases and executable modules that are to be linked and loaded to create the job. Flags to be passed to the linker and loader are also specified in the workspace ***flag*** statement.

set and set-form statements

The ***set*** statement contains a ***set-form*** table which names all database sets and the file names (modules) which contain them. The purpose of these statements is to declare all databases which will be used in the workspace, and include the location where they can be found.

```
(set
   (set-form
      (:type     :name            :module)
      (pinset    d1Mx4_pins       "d1Mx4pins.pindb")
      (chanset   d1Mx4j_chans     "d1Mx4j.chandb")
      (specset   nom_levs         "levels.specdb")
      (specset   timing_70ns      "timing.specdb")
      (specset   timing_nominal   "timing.specdb")
      (eqset     levels_eqns      "pin_levels.leveldb")
      (eqset     simple_wave_eqns "simple_wave_eqns.wavedb")
      (mpgset    dram_patterns    "dram_patterns.svmdb")
      (dgenset   pattern_data     "datagen.dgendb")
      (srcset    srcset           "dram_src.srcdb")
      (mutset    d1Mx4            "dx4.mutdb")
      (startset  march            "patterns.startdb")
   )
)
```

flag and flag-form statements

The ***flag*** statement contains a ***flag-form*** table that specifies the flags to be passed to the standard linker (ld) and the Teradyne linker (tld). Recommended tldflags include -m, and recommended ldflags include -g. The flag objects are given names so they can be included in the job.

```
(flag
      (flag-form
            (:name      :elements   :type)
            (tflags     "-m"        tldflags)
            (lflags     "-g"        ldflags)
      )
)
```

group and group-form statements

The ***group*** and ***group-form*** statements create higher level objects that consist of multiple things. For example, a level object consists of an equation set (eqset) and a specification set (specset). The job is also defined as a group object, since it consists of multiple databases and executable modules. Typically databases and modules are first grouped together to create objects (waveform, level, pattern), and these objects are then grouped to create a job.

```
(group
      (group-form
      (:type            :name       :elements)
      (waveform         W1          (...) )
      (level            L1          ....)
      (pattern          P1          ....)
      (job              Jobname     (
                                    L1
                                    W1
                                    P1
                                    chanset
                                    pinset
                                    "main.o"
                                    )
      )
)
```

Waveforms are created by grouping a waveform equation (eqset) with a specification set (specset):

```
(:type      :name                   :elements)
(waveform   wave_timing_70ns
                                    (
                                    timing_70ns
                                    spec_wave_eqns_1
                                    )
)
```

Levels are created by grouping a level equation (eqset) with a specification set (specset):

```
(:type      :name                   :elements)
(level      nominal_levels
                                    (
                                    nom_levs
                                    levels_eqns
                                    )
)
```

Patterns are created by grouping Pattern control (mpgset), Data generator (dgenset), Source select (srcset), Pattern start conditions (startset), and a Memory under test description (mutset).

```
(:type      :name       :elements)
(pattern    march_patt
                        (
                        march               ;startset
                        dram_patterns       ;mpgset
                        pattern_data        ;dgenset
                        srcset              ;srcset
                        d1Mx4               ;mutset
                        )
)
```

A test program (job) is created by grouping any of the databases and objects together.

```
(:type      :name       :elements)
(job        Dx4J
                        (
                        wave_timing_70ns; waveform
                        nominal_levels      ; level
                        march_patt          ; pattern
                        d1Mx4_pins          ; pinset
                        d1Mx4j_chans        ; chanset
                        "dx4.insadr"        ; insadr
                        "/j997/templates/libtemplates.o"
                        "/j997/etc/exec_instances.insadr"
                        "/j997/lib/libexec_templates.o"
                        "main.o"            ; flow
                        tflags
                        lflags
                        )
)
```

Test programs that use templates will always contain the template library object module (for example, /j997/templates/libtemplates.o). For software version A2.00, the executive instances and templates for summary and datalog also need to be specified (/j997/etc/exec_instances.insadr and /j997/lib/libexec_templates.o). All test programs must contain flag objects, a pinset, a chanset, and an executable module (that contains job_main function).

Procedure for Creating Workspace Database

Use the following procedure to create a new workspace database for a test program. This procedure will utilize all of the databases and modules in a specific directory to create a job.

1) Make sure all required databases and modules exist in a single directory. Change the current working directory to that place.

```
% cd <jobdir>
```

2) Run the applications script ***build_workspace*** to create a starting workspace ADR file. The script will look at the contents of the ADR databases and the source code modules to determine what to put into the workspace. It will create a file Workspace.wspadr in the current working directory.

```
% /j997/demo/scripts/build_workspace
```

3) Edit the file Workspace.wspadr to make it suitable for the desired test program. Correctly name all level, waveform, pattern, and job objects that are contained in the group-form table. It is recommended to use the ADRedit tool for this task. Make sure all objects in the workspace have unique names. This is a requirement.

4) Translate the Workspace.wspadr file to create the Workspace.wspdb file. Resolve any translation errors by editing the ADR file, then attempting translation again.

```
% adr900 Workspace.wspadr
```

5) Make note of all waveform objects, level objects and pattern objects that have been created and defined in the workspace. The names of these objects will be specified in the test program and instance databases.

Once the workspace database has been created, changes or additions to the test program will most likely affect the workspace. This workspace ADR file will be edited frequently as a test program is being developed. For example, a new pattern must be defined in the workspace if the test program will use it.

Workspace Database Example: 1Mx4 DRAM Program

```
; Workspace for J990 test program named "Dx4J"

(set
    (set-form
           (:type       :name              :module)
           (pinset      d1Mx4_pins         "d1Mx4pins.pindb")
           (chanset     d1Mx4j_chans       "d1Mx4j.chandb")
           (specset     nom_levs           "levels.specdb")
           (specset     spec_levs          "levels.specdb")
           (specset     leakage_levs       "levels.specdb")
           (specset     power_off_levs     "levels.specdb")
           (specset     timing_70ns        "timing.specdb")
           (specset     timing_80ns        "timing.specdb")
           (specset     timing_nominal     "timing.specdb")
           (eqset       levels_eqns        "pin_levels.leveldb")
           (eqset       simple_wave_eqns   "simple_wave_eqns.wavedb")
           (eqset       spec_wave_eqns_1   "wave_eqns.wavedb")
           (mpgset      dram_patterns      "dram_patterns.svmdb")
           (dgenset     pattern_data       "datagen.dgendb")
           (srcset      srcset             "dram_src.srcdb")
           (mutset      d1Mx4              "dx4.mutdb")
           (startset    march              "patterns.startdb")
           (startset    scan               "patterns.startdb")
           (startset    addrcompl          "patterns.startdb")
           (startset    vcc_bump           "patterns.startdb")
           (startset    read_write_loop    "patterns.startdb")
           (startset    refresh_loop       "patterns.startdb")
           (startset    refresh_test       "patterns.startdb")
    )
)

(flag
    (flag-form
           (:name :elements :type)
           (tflags  "-m"     tldflags)
           (lflags  "-g"     ldflags)
    )
)

(group
    (default :comment nil)
    (group-form
        (:type      :name             :elements)

        (waveform   wave_timing_70ns
                                      (
                                      timing_70ns
                                      spec_wave_eqns_1
                                      )
        )
```

```
(waveform   wave_timing_80ns
                              (
                              timing_80ns
                              spec_wave_eqns_1
                              )
)

(waveform   wave_timing_nominal
                              (
                              timing_nominal
                              simple_wave_eqns
                              )
)

(level      nominal_levels
                              (
                              nom_levs
                              levels_eqns
                              )
)

(level      spec_levels
                              (
                              spec_levs
                              levels_eqns
                              )
)

(level      leakage_levels
                              (
                              leakage_levs
                              levels_eqns
                              )
)

(level      power_off_levels
                              (
                              power_off_levs
                              levels_eqns
                              )
)
(pattern    march_patt
                              (
                              march             ;startset
                              dram_patterns     ;mpgset
                              pattern_data      ;dgenset
                              srcset            ;srcset
                              d1Mx4             ;mutset
                              )
)
```

```
(pattern    scan_patt
                        (
                        scan            ;startset
                        dram_patterns   ;mpgset
                        pattern_data    ;dgenset
                        srcset          ;srcset
                        d1Mx4           ;mutset
                        )
)
(pattern    addrcompl_patt
                        (
                        addrcompl       ;startset
                        dram_patterns   ;mpgset
                        pattern_data    ;dgenset
                        srcset          ;srcset
                        d1Mx4           ;mutset
                        )
)
(pattern    vcc_bump_patt
                        (
                        vcc_bump        ;startset
                        dram_patterns   ;mpgset
                        pattern_data    ;dgenset
                        srcset          ;srcset
                        d1Mx4           ;mutset
                        )
)
(pattern    read_write_loop_patt
                        (
                        read_write_loop ;startset
                        dram_patterns   ;mpgset
                        pattern_data    ;dgenset
                        srcset          ;srcset
                        d1Mx4           ;mutset
                        )
)
(pattern    refresh_loop_patt
                        (
                        refresh_loop    ;startset
                        dram_patterns   ;mpgset
                        pattern_data    ;dgenset
                        srcset          ;srcset
                        d1Mx4           ;mutset
                        )
)
(pattern    refresh_test_patt
                        (
                        refresh_test    ;startset
                        dram_patterns   ;mpgset
                        pattern_data    ;dgenset
                        srcset          ;srcset
                        d1Mx4           ;mutset
                        )
)
```

```
        (job        Dx4J
                                    (
                                    wave_timing_70ns  ; waveform
                                    wave_timing_80ns  ; waveform
                                    wave_timing_nominal; waveform
                                    nominal_levels    ; level
                                    spec_levels       ; level
                                    leakage_levels    ; level
                                    power_off_levels; level
                                    min_vcc_nom_levels; level
                                    march_patt        ; pattern
                                    scan_patt         ; pattern
                                    addrcompl_patt    ; pattern
                                    vcc_bump_patt     ; pattern
                                    read_write_loop_patt; pattern
                                    refresh_loop_patt; pattern
                                    refresh_test_patt; pattern
                                    d1Mx4_pins        ; pinset
                                    d1Mx4j_chans      ; chanset
                                    "dx4.insadr"      ; insadr
                                    "/j997/templates/libtemplates.o"
                                    "/j997/etc/exec_instances.insadr"
                                    "/j997/lib/libexec_templates.o"
                                    "main.o"          ; flow
                                    "adjust_spec.o"
                                    tflags
                                    lflags
                                    )
        )
    )
)

; end of workspace adr file
```

Chapter 1.11: Creating Instance Database Files

Objective: ***Create .insadr file***
Reference: ***Template Programming (Section 3 of this manual)***
Template Reference Manual
ADR900 Reference Manual
IG900+ Tools Reference Manual

Introduction

Teradyne provides several test ***templates***, which are sets of instructions for performing specific tasks or tests. The templates are executable test code, without the needed parameters or test conditions. When using the test templates, the parameters for the templates are specified by the programmer to create ***instances***. Instances are, therefore, fully specified tests. These instances are executed in the test program to generate test results. These sum of these tests constitute the *program flow*.

The databases created in the previous chapters contain test conditions. These databases will be specified as parameters for the templates. A single template or instance can be used multiple times within a test program.

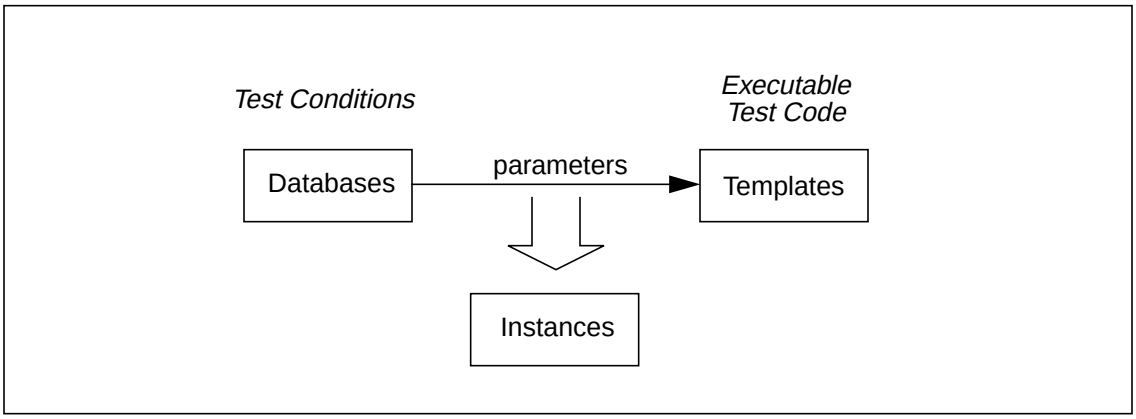


Figure 1.11-1 Test Conditions Databases (specdb, leveldb, svmdb, etc.) and Executable Code (templates) Combined to Create Instances

Creating Instance Databases

Instances are usually created with ***Insted*** (the instance editor) and are stored in instance databases. Instance databases are stored in ASCII form, and are contained in files with the ***.insadr*** extension.

Since the instance database will be created by Insted, the procedure for creating the ***.insadr*** file will be different from creating the test conditions databases. In addition, the test conditions will be used as parameters in the instance database.

Once the instance database file has been created, it will be included in the workspace database for the test program.

The basic sequence for creating test instances is:

- ❒ Decide what tests are to be done, and list the conditions and databases which will be needed for each test.
- ❒ Review the template reference documentation and the template code itself to select the templates to be used for each test.
- ❒ Use Insted to specify the parameters to be used with the templates, creating the Instance database file (.insadr file). If the pins, levels, waveforms, patterns, and workspace have already been created and translated, insted can read them to show which can be used as instance parameters.
- ❒ Verify the Instance database file using the ***adr900*** translator. (The *.insdb* file can then be deleted, as the ***.insadr*** file is the one used in the job).
- ❒ Add the instance database file and the template library to the workspace ADR file.

Selecting Templates

For a comprehensive description of all templates, refer to the *Templates Reference Manual*. You may also view the template code by looking at the .t files located in the templates directory (/j99x/templates[1]). Use the following summary as a general guide to using some of the basic templates.

background_Template

This template is used to alter the background or starting conditon of a test program from the Executive's default state. This template is optional. Because it can save a significant amount of processing time, Teradyne recommends its inclusion in every job. If none of the above tasks are needed in a test program, it can be omitted.

Sets up parameters for the entire job program by performing the following tasks:

- ❒ Enabling and/or disable utility bits
- ❒ Disabling comparators on pins (for debug)
- ❒ Switching "open" the normal state for a device pin
- ❒ Programming of AT900 "on" conditions

change_conditions_Template

The change conditions template can be used to provide basic functional test capability (at a very primitive level); however, its main use is to change the current waveform, level, pattern, calset, or test number interactively in Insted. This template is typically used to change conditions inherited by shmoo.

To use change conditions, put an "empty" instance in the .insadr file. (All parameters of the template are optional). Do not execute the instance in your normal job flow. Then, at a trap, use Insted to open the change condition instance, and select whatever parameter desired. Execute the instance, and shmoo will run under the new conditions.

corner_list_Template

This template specifies a series of functional tests, called *corners*. Each test is fully specified: the pattern, waveform, levels, and names are individually entered in the template. The template provides many options for selecting which corner(s) out of the list are to be executed by an instance. It is particularly useful for interactive use in conjunction with Insted.

func_Template

Implements a functional test using a list of one or more patterns. The template sets up the specified waveform and level, then runs the specified patterns. Allows for the option of executing a single test or a test after each functional pattern. Recommended for functional testing where a device pass/fail is needed.

1. Substitute for **x** the name of your system as in j99**4** or j99**7**.

fvm_setup_Template

This template specifies fvm conditions. It allows the programming of the patgen stop mode. These are a choice among:

- ❒ stop on stop opcode only (run all patterns to completion)
- ❒ stop on stop opcode OR CR latch (stop patgen if all devices fail)
- ❒ stop on stop opcode OR FVM stop (stop patgen when FVM is full))
- ❒ stop on stop opcode OR FVM stop OR CR latch
- ❒ stop on stack error (this is ina ddition to above entries)

Various FVM parameters can also be set. These parameters include patgen pipeline length, FVM storage mode (store fails only, or all vectors), preview and postview (number of vectors to be stored before and after a failing event), and others.

pin_pmu_Template

Perform one or two DC parametric tests on one or more pins using the pin PMUs. The template can provide a pattern run before and during a test. Often used for device pin leakage tests.

pin_pmu_alternate_Template

Same as pin_pmu_Template, except provides for the specification of two lists of pins for alternating high and low voltages. Can be used for more complex leakage tests.

ivs_Template

Performs a voltage source current monitor test, allowing for selection of pattern burst before and during a test, and specification of pins to be grounded or open. This template is often used for Icc operating tests.

ivs_pmu_Template

Performs a voltage source current test by using the PMU connected to a VS pin. A pattern can be run before and during the test. This template is often used for Icc standby tests, where the extra resolution of the PMU is desired.

datasheet_Template

Performs an Adjust-Until on a spec in an attempt to find the pass/fail boundary. Used for parameter characterization tests. The spec can be characterized for a list of reference values such as Vcc.

Using Insted

The Instance Editor (Insted) provides editing capability for instances, by accepting templates, then providing a fill-in-the-blank method for specifying parameters. It checks to make sure all required parameters have been specified, then creates the instance database file.

The view button in the main display window (Insted Manager Display) summons the templates list subwindow and the tree display subwindows. Either subwindow can be used to view the existing instances in the database. The tree subwindow does not illustrate flow, but does show the relationship between templates and their instances.

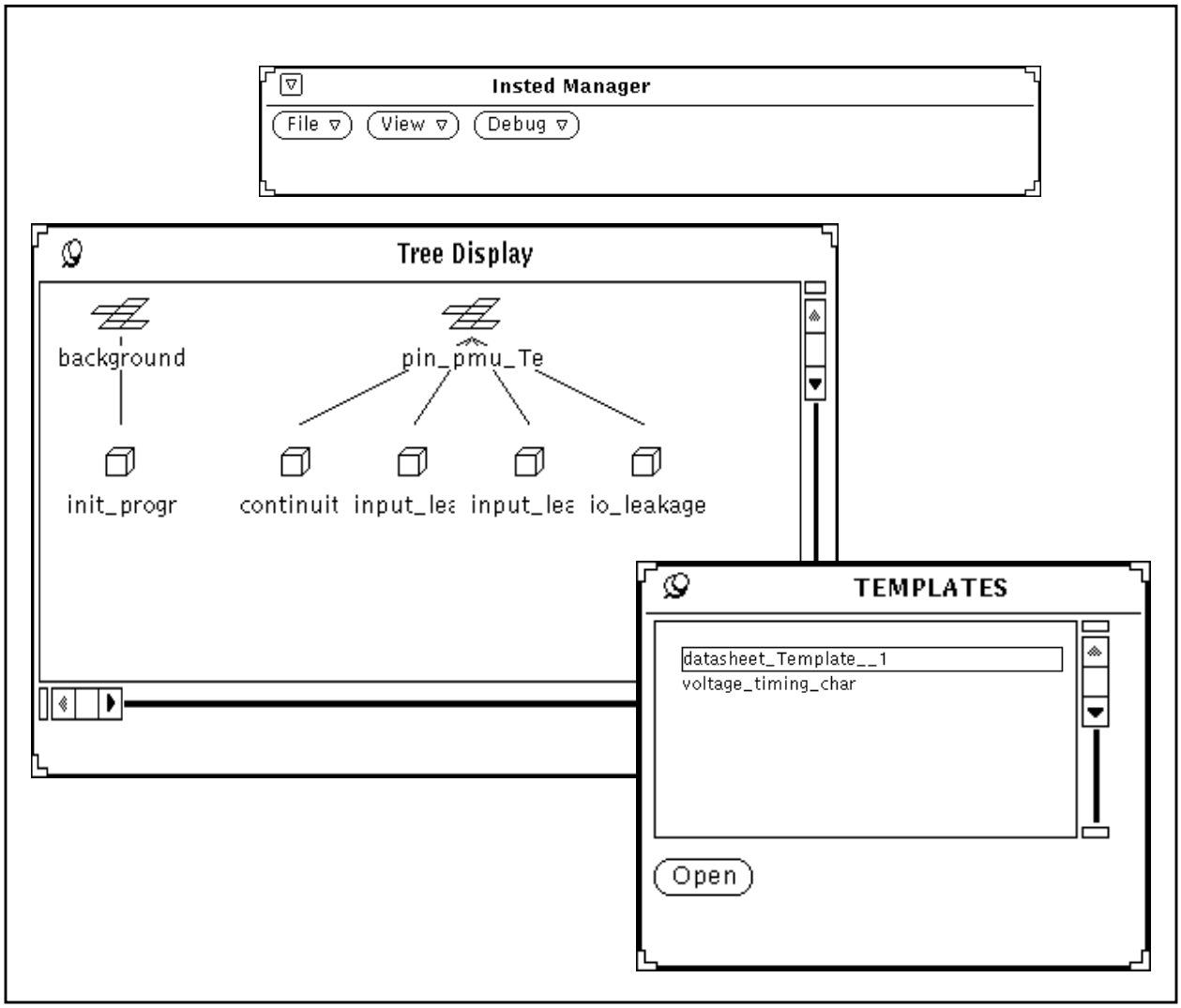


Figure 1.11-2 Insted Main Display Window, Template List, and Tree Display

To edit a particular instance, the Editor Panel is used. It shows a blank line for each available template variable. Parameters are specified by entering values into the blanks.

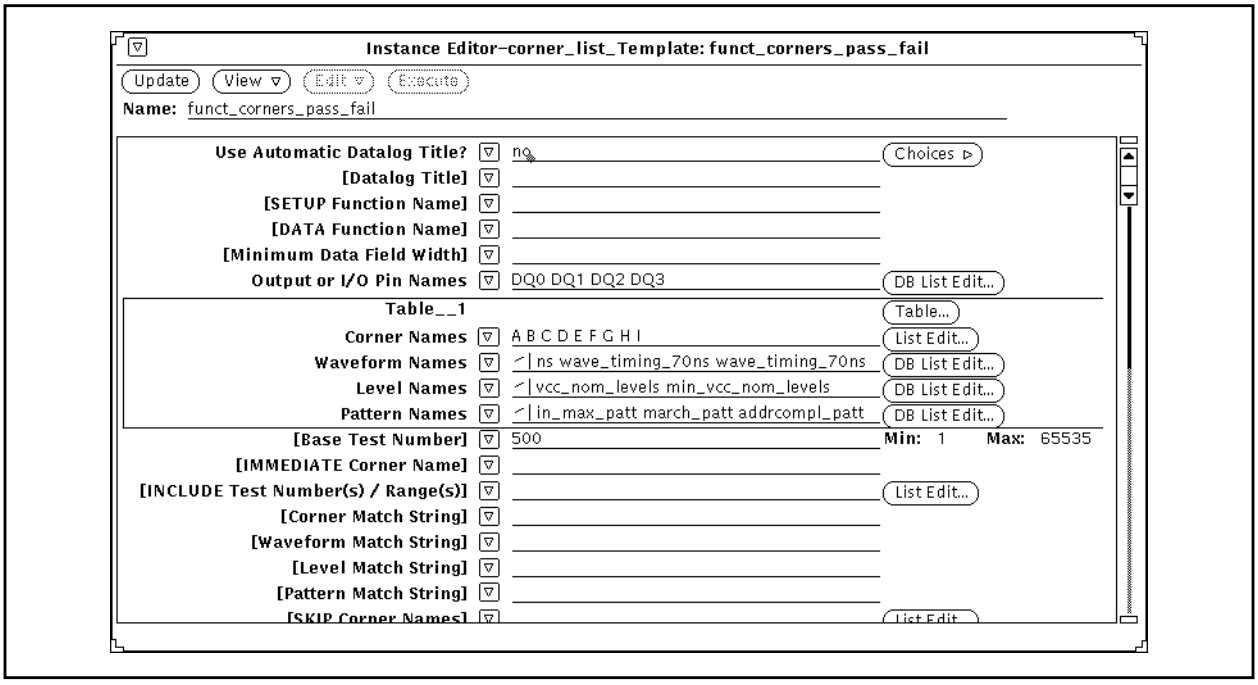


Figure 1.11-3 Instance Editor Panel

Procedure for Creating Instances

The typical procedure for creating new instances involves including template description files, creating an instance from it, then editing the parameters for that instance. Repeat that procedure until all instances are created, then saving the instances to an instance database file (.insadr).

1) Bring up Insted using the Openwin menu or entering insted into any shelltool or cmdtool. Insted will appear as an empty tool.

2) Select a workspace. To select a workspace, select the **File** button, then **Load**. The load panel appears. Select **Object Type** and switch workspace to pick a new workspace. Select your workspace (created in Chapter 1.10); the wspdb file must exist. Now all Insted buttons will be operational.

3) Include the templates that are needed from the Teradyne template directory (for example, /j997/templates).

   - ❒ In the Insted Manager, selecting the **Include** option from the **File** menu.
   - ❒ In the **Include** window, use the menu button to select the **Template Definition File** option from the **Object Type** menu.
   - ❒ Next to **Directory**, input the correct directory and path to the location of the template files. (for example, /j997/templates).
   - ❒ After list of templates appears in the bottom of the screen, highlight a template and press the **Include** button.

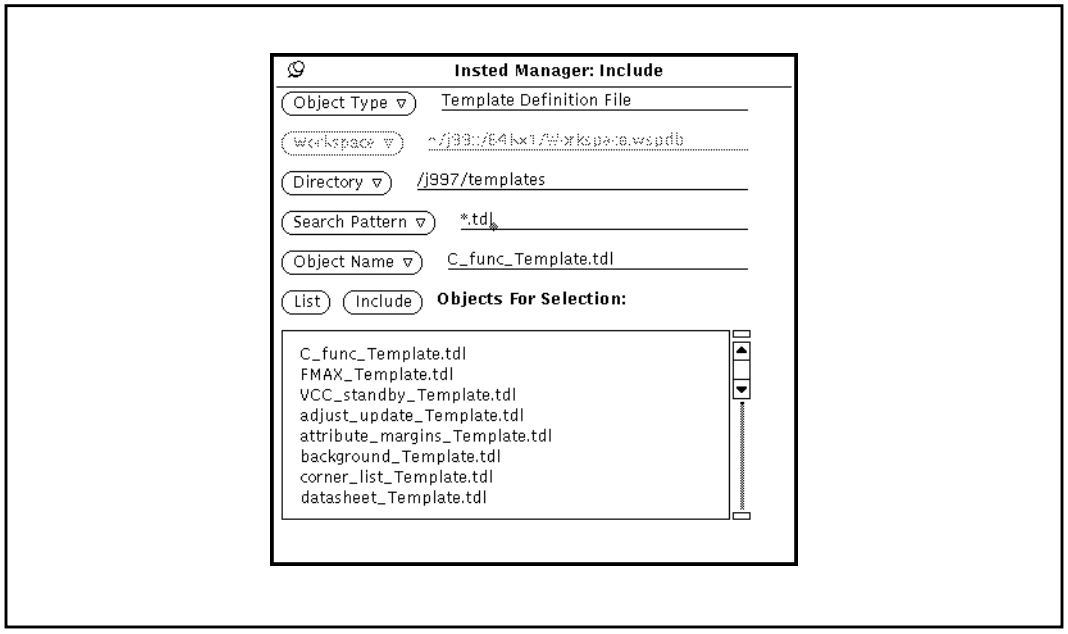


Figure 1.11-4 Including a Template into the Editor

- ❒ Include all the templates which will be needed, then close the include panel. Each included template will appear in the tree display or template list.

4) Select a template for editing by double-clicking the left mouse button (select button) on a template icon or list or by using the **Open** button. A instance will automatically be created from the selected template, and the instance Editor Panel will appear.

   - ❒ Change the existing text next to the **Name** field to the desired name of the instance. Press return. The instance will be renamed to the specified name.

5) Edit the parameters for the instance by filling in values next to the blanks.

   ❒ Values can be filled in by entering text directly from the keyboard, or using the pull down and pop-up panels next to each field. Entering a return after each field will prompt Insted to accept the parameter and verify it.

   - optional parameters will be surrounded by brackets [ ] and are displayed in lighter color text. Some parameters are shown as optional but may be required based on the value of other parameters. If these parameters are not filled in, they will generate runtime errors.,

   - all other parameters are required to have a value. The parameters are:

     integers
     character strings
     doubles (these are used for values such as voltages and nanoseconds; do not include the actual unit after the number.)
     IG900+ objects (like levels, waveforms, patterns, or pins)
     names of functions (These functions must be typed as
         extern "c" int abc();)

   ❒ Repeat this step until all instances have been created. The tree or list will be updated every time a new instance is created, showing the templates and the instances.

6) Save the instances to an Instance Database file. This will create the .insadr file that will be used by the test program.

   ❒ Select **Save as...** from the **File** button.

   ❒ Select **Instance to a File** from the **Object Type** menu.

   ❒ Next to **Directory**, enter the correct path and directory into which the file will be saved. This may have already been set when loading the workspace. If there are other instance files in the directory, they will appear at the bottom of the screen.

   ❒ Select one of the listed files or enter a new file name next to **Object Name** and press the **Save As** button. If an existing file name is selected, the existing file will be overwritten (after confirmation). If a new name is entered, then a new file will be generated.

An instance database file will be an ADR file which contains a **define-instance** statement for each instance that was created. This file can be edited using Insted or a standard text editor like **vi**.

When creating new instances, Insted should be used because it can show objects defined in the workspace. When a single instance parameter is to be modified in the database file, it may be quicker to just edit the **.insadr** file with **vi**. A sample **define-instance** statement from an Instance database file is shown below. This is only one portion of the .insadr file.

```
(define-instance funct_nominal func_Template
      (value patterns addrcompl_patt march_patt )
      (value stop_on_first_fail 0 :choice "no")
      (value dotests 1 :choice "yes")
      (value autotitle 0 :choice "no")
      (value tnum )
      (value trap )
      (value skip )
      (value driveroff )
      (value driver0 )
      (value driver1 )
      (value calset )
      (value waves wave_timing_nominal)
      (value levels nominal_levels)
      (value title "functional test - nominal conditions")
      (value patstop )
      (value patstart )
      (value postbody )
      (value prebody )
      (value posttest )
      (value pretest )
)
```

7) Specify the Instance Database file in the job group in the workspace database.

   ❒ Use a text editor to edit the **.wspadr** file, and proceed to the group object for the job.

   - Add the instance database file name as part of the job object, enclosing the name in quotes.

   - Add the template library file name as part of the job object, enclosing the name in quotes.
     (Example: /j997/templates/libtemplates.o)

A sample job group (from a .wspadr file) appears below.

```
(job  Dx4J
            (
            wave_timing_70ns        ; waveform
            wave_timing_80ns        ; waveform
            wave_timing_nominal     ; waveform
            nominal_levels          ; level
            spec_levels             ; level
            march_patt              ; pattern
            scan_patt               ; pattern
            addrcompl_patt          ; pattern
            d256Kx4_pins            ; pinset
            d256Kx4j_chans          ; chanset
            "d256Kx4.insadr"        ; INSADR ****
            "/j997/templates/libtemplates.o"
            "main.o"                ; flow
            tflags
            lflags
            )
)
```

Chapter 1.12: Creating a Flow Control Module

Objective: ***Create main.t and main.o files***
Reference: ***AT900 Reference Manual***

Introduction

Once all the test conditions and workspace databases have been created, the last program module to be created is the flow control. The flow control module is written in AT900 language, and controls the execution of tests and instances.

At least one flow control module (file) must be created for a job. There will be one job_main function defined in the file, and this function contains the executable code which will be run when the test program is run. The flow control module can contain any combination of AT900 and C++ programming statements.

The flow control module contains the main flow of the program, and is typically named main.t, but any acceptable UNIX name can be used. The main flow often contains program documentation and revision history notes.

The flow control module is created with any standard text editor. It is then compiled by the AT900 compiler, generating an object module which is linked into the program.

Contents of Flow Control Module

There are only two requirements for the main flow control module.

1) It must include the standard Teradyne header file teradyne.h.

```
#include <teradyne.h>
```

This header file defines structures and macros which can be used in any AT900 language module.

2) It must contain the function job_main.

```
job_main() {
   <... flow code ...>
}
```

Inside the job_main function, the programmer can execute any AT900 statement, use most AT900 macros (including execution of test instances), and call other C or C++ functions.

The main() function typically seen in C and C++ programs is reserved and controlled by Teradyne. This code is executed when the test program is loaded. It is the user function *job_main* which gets executed when the test program is run.

Typically, the flow control module executes instances and makes branching decisions based on their results (including sorting and binning).

Executing Instances

Instances are first imported using the *import* statement outside of the *job_main* function. The *EXECUTE* macro is used inside a function to actually execute the instances. Instances are defined in the instance database and essentially represent one call to a test template. The instance will automatically be assigned a result.

Branching and Flow Control Based on Test Results

The result from the instance can be used to affect the program flow. This is accomplished with the *FAIL_DO* and *PASS_DO* macros. These macros are used to execute a block of code on those devices which passed or failed an instance. Program statements inside of a FAIL_DO block will be executed on those sites (devices) which failed the specified instance. Program statements inside a PASS_DO block will be executed on those sites which passed.

```
#include <teradyne.h>
import Instance funct_test;
job_main()
{
      EXECUTE(funct_test);
      FAIL_DO(funct_test) {
            ... statements executed on devices which FAIL ...
      }
      PASS_DO(funct_test) {
            ... statements executed on devices which PASS ...
      }
```

The statements inside the FAIL_DO and PASS_DO blocks are conditional. If all devices pass, then the code within the FAIL_DO will not be executed.

The parallel job executive automatically keeps track of those devices which passed or failed an instance. It maintains a *current siteset* which is a list of sites that are considered active. For example, if a test program had been set up to run on sites 0, 1, 2, and 3 on test head 0, the current siteset would be *h0 0..3*. If an instance was executed and site 2 failed, then the FAIL_DO code would be executed with only site 2 active. Site 2 (*h0 2*) would be the current siteset while the code within the FAIL_DO was executed. Sites 0, 1, and 3 (*h0 0..1,3*) would be the current siteset while a the PASS_DO block was executed. Because the parallel job exec keeps track of active, passing, and failing sites, this reduces the amount of logic that must be coded into a test program.

FAIL_DO and PASS_DO statements can be nested. Instances can be executed and FAIL_DO and PASS_DO blocks can be placed inside FAIL_DO and PASS_DO.

Compiling Flow Files

Once the flow file has been edited, it must be compiled with the AT900 compiler, which will generate an object module.

```
% at900 main.t
```

Once the object module exists, it must be specified in the workspace ADR file (.wspadr). It must be listed as part of the job object.

Example Flow File

```
/* File: main.t */

#include <teradyne.h>

int bin(int);

import Instance
     basic_functional,
     continuity,
     leakage,
     icc_standby,
     icc_operating,
     funct_spec_70ns,
     funct_spec_80ns;

/* Sort/Bin Number Definitions */

#define FAIL_BFUNC   14
#define FAIL_CONTIN  13
#define FAIL_LEAKAGE 12
#define FAIL_ICC_STDBY11
#define FAIL_ICC_OPER10
#define FAIL_SPEC_FUNC9

#define PASS_SPEC_80 2
#define PASS_SPEC_70 1
```

```
/*
 *   Function: job_main()
 *
 *  Job main flow.  After each type of test is performed, binning is
 *  immediately performed if the any test of that type failed.  A speed
 *  sort loop is performed at the end.
*/

job_main()
{
    EXECUTE(basic_functional);
    FAIL_DO(basic_functional) {
            EXECUTE(continuity);
            FAIL_DO(continuity) {
                    bin(FAIL_CONTIN);
            }
            PASS_DO(continuity) {
                    bin(FAIL_BFUNC);
            }
    }

    EXECUTE(leakage);
    FAIL_DO(leakage) {
            bin(FAIL_LEAKAGE);
    }

    EXECUTE(icc_standby);
    FAIL_DO(icc_standby) {
            bin(FAIL_ICC_STDBY);
    }

    EXECUTE(icc_operating);
    FAIL_DO(icc_operating) {
            bin(FAIL_ICC_OPER);
    }

    EXECUTE(funct_spec_70ns);
    FAIL_DO(funct_spec_70ns) {
            EXECUTE(funct_spec_80ns);
            FAIL_DO(funct_spec_80ns) {
                    bin(FAIL_SPEC_FUNC);
            }
            PASS_DO(funct_spec_80ns) {
                    bin(PASS_SPEC_80);
            }
    }
    PASS_DO(funct_spec_70ns) {
            bin(PASS_SPEC_70);
    }

    return(0);
}
```

```
/*
 * Function: bin();
*/

bin(int bin_num)
{
     printf("%s: Sort/Bin %d\n",readsys(current_siteset),bin_num);
     if (bin_num <= 3) {
            writesys(sort_nokeep,bin_num,TT_BIN_PF_PASS);
     }
     else {
            writesys(sort_nokeep,bin_num,TT_BIN_PF_FAIL);
     }
     return(0);
}
```

Chapter 1.13: Overlay Level and Waveform Files

Objective: ***Create .oldb and .owdb files***
Reference: ***ADR900 Reference Manual***

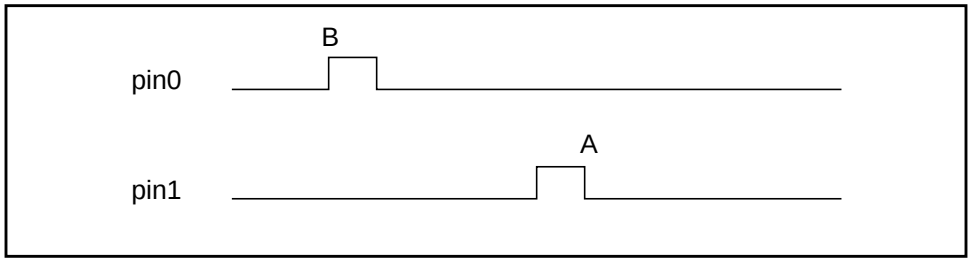
Introduction

Overlay waveforms and overlay levels provide another method for moving edges. Instead of creating an entirely new .wavedb or .leveldb to move an edge, you can use a small .owdb or .oldb file called an overlay.

In addition, overlays can shorten job program generation time and reduce load time by cutting down on file space. For example, overlay waveforms can lower the number of waveform databases that must be created when shmooing several combinations of edges with different dependencies. Overlay levels may be used in the same way.

Another benefit is overlays allow the equations in the original waveform or level to be left unchanged after a session of editing and shmooing the equations used in an overlay.

A main difference between overlay objects and waveform and level objects is overlays contain only a few pins and edges, whereas waveforms and levels require all edges of a pin to be explicitly defined or defaulted.



Procedures for Creating Overlays

The following are methods for creating overlay waveforms and overlay levels.

Creating Overlays with ADR900

This method allows for the most flexibility. It involves creating overlay .db files with ADR900.

1. Define the specsets or use specsets in an existing .specadr file.
2. Create the overlay waveforms or overlay levels.
   a. For overlay waveforms

      Create a .owadr file , which uses .waveadr file syntax, but omits the following:

      - The *(patgen :freq)* statement
      - *:patgen*, *:driver*, *:driver-init*, *:comp*, *:strobe*, *:mach2*, *:scio*, *:chan-mux*, and *:io* in the (*pins-form...)* and (*mcgpins-form...)* statements
      - The (*dvar-form...)* statement in the *(eqset...)* statement
      - The *:fmt* or *:palette* clause in the *equation-form* clause

      Note that equations in an overlay can reference the edges of the background waveform by prepending the edge with dollar sign symbol ($). For example,

      Background waveform: prg.d0@rw_inst = "5ns"

      Overlay waveform: cs.d0@rw_inst = "$prg-tCS"

   b. For overlay levels

      Create a .oladr file that uses .leveladr file syntax, but omits the following:

      - The *(range :level...)* statement
      - The *:power-seq* and *:pause* headings in the *(pins-form...)* statement
      - The (*dvar-form...)* statement in the *(eqset...)* statement
3. Define the overlay objects in the workspace.
   a. If they do not already exist, list the specsets and eqsets in the *set-form* statement. For example:

```
        [...]
        (set
        (set-form
            (:type          :module                 :name    )
            (specset        "FILENAME.SPECDB"     AC_SPECSET_NAME)
            (specset        "FILENAME.SPECDB"     DC_SPECSET_NAME)
            (eqset          "FILENAME.OWDB"
OVERLAY_WAVEFORM_EQSET_NAME)
            (eqset          "FILENAME.OLDB"       OVERLAY_LEVEL_EQSET_NAME)
        [...]
```

   b. Define the overlay waveforms and overlay levels as group objects in the *group-form* statement as shown below.
   c. Add the newly-created overlay objects to the *job* group as shown below.

```
[...]
```

```
(group
    (default :comment "")
    (group-form
        (:type              :name                   :elements)
        (overlay_waveform   OVERLAY_WAVEFORM_NAME (AC_SPECSET_NAME

OVERLAY_WAVEFORM_EQSET_NAME)
        )
        (overlay_level      OVERLAY_LEVEL_NAME    (DC_SPECSET_NAME
                                                   OVERLAY_LEVEL_EQSET_NAME)
        )
        (job                                      JOB_NAME
                                                  (OVERLAY_WAVEFORM_NAME
                                                  OVERLAY_LEVEL_NAME
[...]
    )
)
```

4. Translate the .oladr, .owadr, and .wspadr files to .*db binary files using the **adr900** translator command.

Building Overlays from Existing Specsets and Eqsets

The next two methods are used for creating overlays that are combinations of existing specsets and eqsets. Both can only be done when running a job. The overlays will only exist in memory while a job is being run. After the session has been ended, the overlays that you created will disappear.

Note that the eqsets and specsets used in overlays must be a part of the job group in the workspace. They must also be listed explicitly in the job group, or implicitly through a waveform, level, overlay waveform, or overlay level named in the job group.

Using AT900 Statements

Within a .t file, enter one or both of the following AT900 commands:

```
writesys(select_overlay_waveform, (OVERLAY_WAVEFORM_EQSET_NAME, AC_SPECSET_NAME));
writesys(select_overlay_level, (OVERLAY_LEVEL_EQSET_NAME, DC_SPECSET_NAME));
```

The new overlay will be called

tt_ow_AC_SPECSET_NAME_OVERLAY_WAVEFORM_EQSET_NAME or tt_ol_DC_SPECSET_NAME_OVERLAY_LEVEL_EQSET_NAME.

Note that you may omit the specset name in the previous select statements to dynamically create an overlay containing the specified eqset and the specset in the current waveform or level.

Using Job State

To create new overlays using the IG900+ tool Job State

1. From Jobtool, trap on a test, test type, or instance.
2. Summon Job State from the Jobtool's **Tools** menu.
3. In Job State, click the right mouse button over **Overlay Level** or **Overlay**

Waveform select **Create...** from the pullright menu.

4. In the **Create** window, enter next to **Name** the name of the new overlay.
5. From the **Eq Set/Table** menu, select **List...**.
6. In the **List** window, select an eqset.
7. From the **Spec Set/Table** menu, select **List...**.
8. In the **List** window, select a specset.
9. In the **Create** window, select **Create...**.
10. To select an overlay and view the results:
    a. In Job State's main window, select the new overlay listed in either the **Overlay Waveform** or **Overlay Level** menu and press **Update**.
    b. Select **Results** from the **View** menu.
    c. In the **Results** window, select **Once** (to run the test over again) or **Cycle** (to run the test repeatedly) from the **Restart** menu.

Editing Overlays

Overlays, created by any of the previous methods, can be edited on a value-by-value and equation-by-equation basis with Specsheet and Calcsheet.

1. From Jobtool, trap on a test, test type, or instance.
2. Summon Job State from the Jobtool's **Tools** menu.
3. Select an overlay waveform or overlay level from the **List** window, which can be summoned from either the **Overlay Waveform** or **Overlay Level** menu located in the Job State's main window.
4. Select **Update**.
5. Summon Specsheet or Calcsheet from the Jobtool's **Tools** menu.
6. From either Specsheet or Calcsheet press **Debug**.
7. Change the desired value(s) and equation(s).
8. Press **Update** to update the database and the hardware.
9. If desired, save the changes by using either the **Save** or **Save as** option in the **File** menu.

Procedures for Using Overlays in a Job

There are several ways in which overlays may be applied to a job: within AT900 or through an IG900+ tool.

AT900

Overlays can be implemented within a job by using AT900 statements in the main .t flow file. Any *readsys* or *writesys* statement that uses *waveform* or *level* may be substituted with *overlay_waveform* and *overlay_level*, respectively. The standard rules found in the *AT900 Reference Manual* still apply.

To use overlays with AT900, always begin by inserting the following *import* statement in the .t job flow file:

```
import overlay_waveform OVERLAY_WAVEFORM_NAME;
import overlay_level OVERLAY_LEVEL_NAME;
```

To select an overlay object, use the following AT900 select statements:

```
writesys(select_overlay_waveform, OVERLAY_WAVEFORM_NAME);
writesys(select_overlay_level, OVERLAY_LEVEL_NAME);
```

Similar to selecting a waveform or level, these statements write the current values to the memory structure and to the hardware. Only those edges specified in the database will affect the hardware; items not programmed are not set. You must first select a waveform and then select an overlay waveform, so the edges can be manipulated by the waveform overlay without modification to the background waveform. Overlay levels should be used in the same manner.

No more than one set of a background waveform and an overlay waveform can be active at one time. To select another overlay waveform, you must always select or re-select a background waveform and then select the new overlay waveform. The same is true for levels.

To modify or update portions of the overlay selections in the hardware, use the following AT900 statements:

```
writesys(update_overlay_waveform, OVERLAY_WAVEFORM_NAME);
writesys(update_overlay_level, OVERLAY_LEVEL_NAME);
```

In addition to overlays, you can also update overlay waveforms and overlay levels. Note that the location where the statement transfers or writes data (i.e., from the database to the hardware, to the hardware only, or to the database only) affects test results. For example, if the waveform named tight_timing had the values t1 = 10ns, t2 = 12ns, and t3 = 14 ns; the overlay named loose_timing had a single edge value of t2 = 16 ns; and the following commands were entered:

```
writesys(select_waveform, tight_timing);          //TRANSFERS FROM DATABASE TO
                                                    HARDWARE
writesys(select_overlay_waveform, loose_timing);  //TRANSFERS FROM DATABASE TO
                                                    HARDWARE
writesys(spec_value, t2, tight_timing,18ns);      //WRITES TO DATABASE ONLY
writesys(update_overlay_waveform, tight_timing);  //WRITES TO HARDWARE ONLY
run FUNCTION
```

then the function would use 18 ns for a timing value. If, however, the following were run

```
writesys(select_waveform, tight_timing);          //TRANSFERS FROM DATABASE TO
                                                    HARDWARE
writesys(select_overlay_waveform, loose_timing);  //TRANSFERS FROM DATABASE TO
                                                    HARDWARE
writesys(spec_value, t2, tight_timing,18ns);      //WRITES TO DATABASE ONLY
writesys(update_waveform, tight_timing);          //WRITES TO DATABASE ONLY
run FUNCTION
```

then t2 would be 16 ns.

A NULL reloads the original database values. For example, if the following were run

```
writesys(select_waveform, tight_timing);          //TRANSFERS FROM DATABASE TO
                                                    HARDWARE
writesys(select_overlay_waveform, loose_timing);  //TRANSFERS FROM DATABASE TO
                                                    HARDWARE
writesys(spec_value, t2, tight_timing,18ns);      //WRITES TO DATABASE ONLY
writesys(update_overlay_waveform, NULL);          //WRITES TO HARDWARE ONLY
run FUNCTION
```

then t2 would be 16 ns. (Note that updates using the NULL argument always affect the last update or selection.) If, however, the following were run

```
writesys(select_waveform tight_timing);           //TRANSFERS FROM DATABASE TO
                                                    HARDWARE
writesys(select_overlay_waveform loose_timing);   //TRANSFERS FROM DATABASE TO
                                                    HARDWARE
writesys(spec_value, t2, tight_timing,18ns);      //WRITES TO DATABASE ONLY
writesys(update_waveform, NULL);                  //WRITES TO DATABASE ONLY
run FUNCTION
```

then t2 would be 16 ns.

Interactive Usage of Overlays

Overlays can be applied using the IG900+ tool Job State, which involves the following steps:

1. From Jobtool, trap on a test, test type, or instance.
2. From Jobtool, summon Job State from the **Tools** menu.
3. In Job State's main window, select an overlay listed in either the **Overlay Waveform** or **Overlay Level** menu and press **Update**.

Chapter 2.1: Programmer's Guide to the Hardware

Introduction

This section provides conceptual descriptions of the system hardware from a programming standpoint. Block diagrams of the hardware are included, with references to those pieces of software used to program the hardware. New J990 Series programmers will find this section useful for gaining knowledge about test system capabilities.

The J990 Series consists of three system types: the J997, the J994, and the J993. The J997 has the highest performance in the series. It has the following features:

- ❒ 100/200 MHz pattern generator (***patgen***) operation
- ❒ single, dual, or dual asynchronous pattern generator operation
  - two data generators per pattern generator
- ❒ 16X, 16Y address generation
- ❒ 1 or 2 round test heads
- ❒ up to 16 device test sites in parallel per head (up to 32 sites total)
- ❒ 72 IO channels per test head
- ❒ 320 Drive-only channels per test head divisible as:
  - 40 drive-only channels per site for up to 8 sites
  - 20 drive-only channels per site for up to 16 sites
- ❒ up to 320 Megabits of Catch RAM for error processing
- ❒ 32 edge sets (timing and format)
- ❒ 1024 pattern control steps
  - 3 pattern control bits per IO channel per pattern location
  - 4 pattern control bits per drive-only channel per pattern location

The overall system block diagram shown in the Figure 2.1 illustrates the basic subsystems, including:

- ❒ Pattern generators
- ❒ Waveform generators (timing and format)
- ❒ IO channel cards
- ❒ Drive-only channel cards
- ❒ Address scramblers
- ❒ Error Processing (Catch RAM and redundancy analyzers)
- ❒ Address-Data Source (ADS) bus

Each of these subsystems will be described in detail in this chapter.

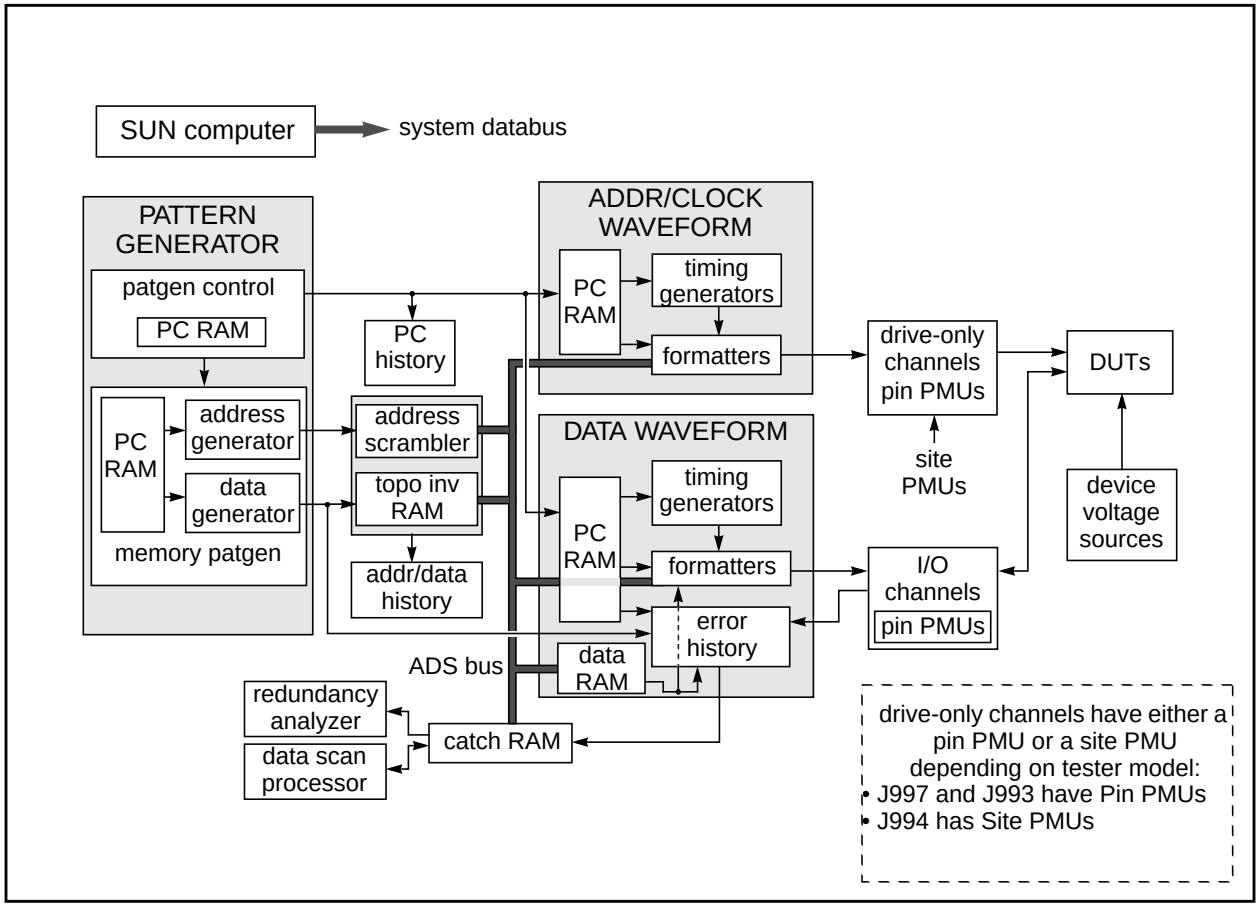


Figure 2.1-1: Overall System Block Diagram

Drive-only channels in the J997 and J993 have pin PMUs, while the drive-only channels in the J994 have a PMU per hardware site.

The **J993** has an architecture similar to the J997 but it is optimized for high-volume wafer applications. It has a frequency of 60/120 MHz and support 72 IOs and 320 address clock channels per test head. Up to 16 sites per test head can be tested on a J993. Another version of the J993, the J993-01, can support 32 sites per test head.

The **J994**, the third member of the series, also shares the same system architecture, but it is optimized for high-volume package testing applications. Unlike the J997 and J993 which have round heads, the J994 has a square test head which can be configured with up to 144 IOs per test head and 640 address/ clock drivers. Two test heads can be installed in a system. Each of these test heads can test up to 32 devices in parallel. The drive-only channels in the J994 utilize a site PMU instead of a pin PMU. The J994 is a 60/120MHz system.

System Frequency Modes

The systems have the flexibility to operate at a standard speed and in a high-speed mode; they can also operate synchronously and asynchronously. These modes include:

- ❒ Normal 100 MHz mode or Normal 60 MHz mode
- ❒ Dual Synchronous 200 MHz mode or Dual Synchronous 120 MHz Mode
- ❒ Dual Asynchronous Mode

Each of these modes is described briefly in the next few pages.

100 MHz Mode

When the system is operating in the 100 MHz mode (J997), two timing and control signal paths are used throughout the system. These two paths (A-Path and B-path) are used together to achieve up to 100 MHz operation of the pattern and waveform generators. Generally, the system is programmed to operate in this mode, as this permits the most flexibility in terms of edge placement from cycle-to-cycle.

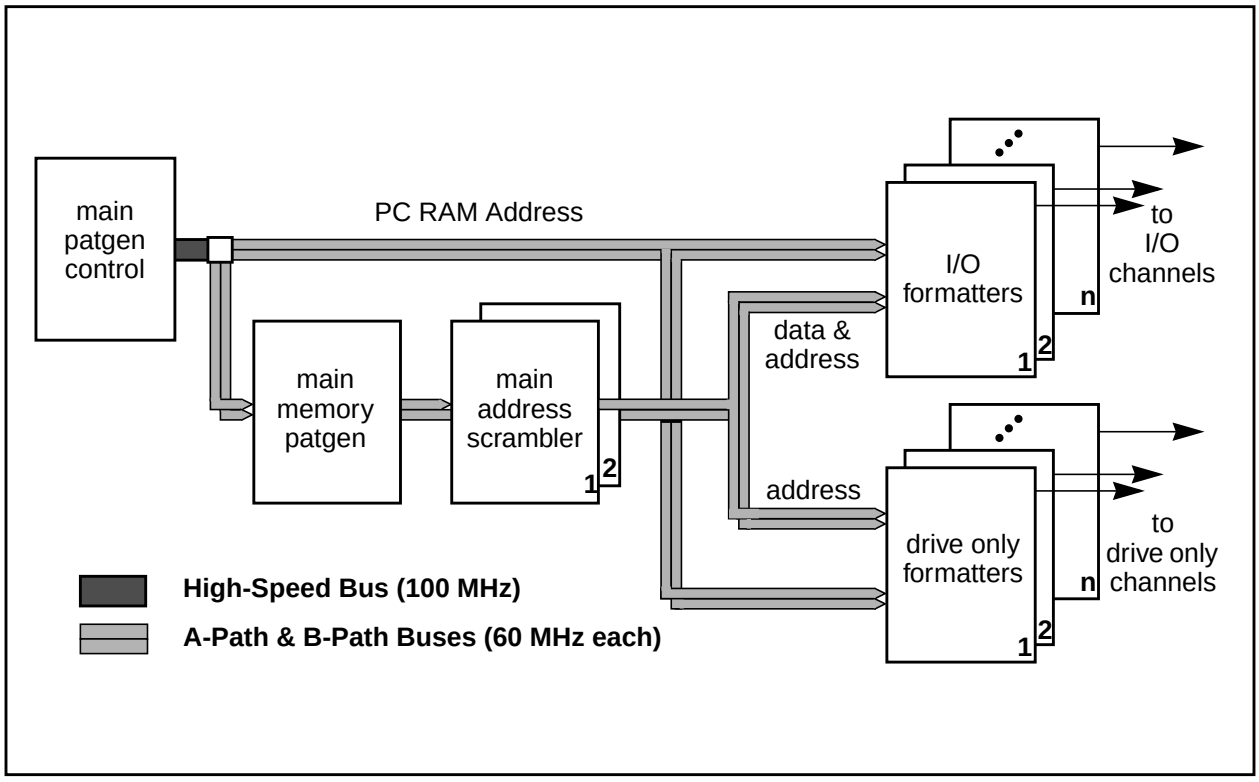


Figure 2.1-2: System Operation in the 100 MHz mode

60 MHz Mode

When the system is run in the slower, 60MHz mode, only the A-path is used. Using the 60 MHz mode (on a J997) results in more timing programming restrictions that must be adhered to, but will assure compatibility with the J994 or J993 test systems, for which this mode is the standard operating mode.

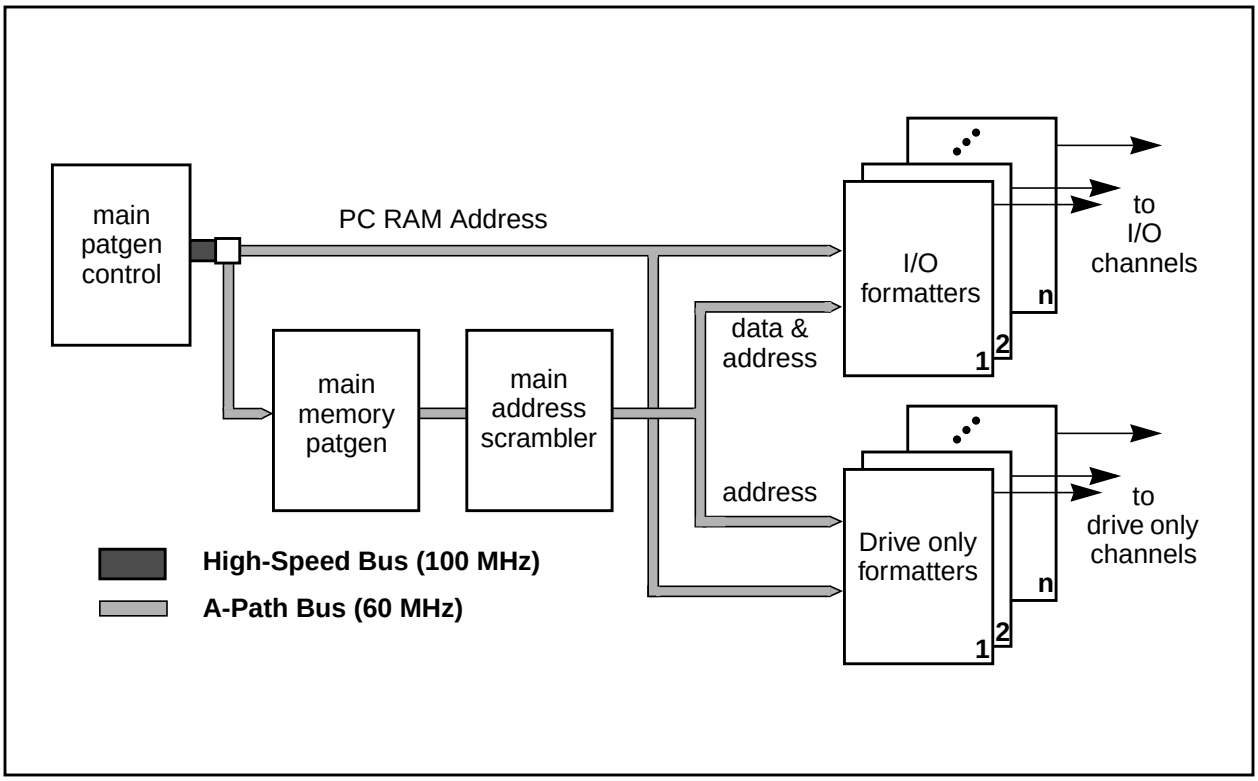


Figure 2.1-3: System Paths in the 60 MHz mode

Dual Synchronous Patgen Operation (200 MHz mode)

When operating at speeds higher than 100 MHz, two memory pattern generators operate in parallel, both receiving A-Path and B-Path information. The A-Path and B-Path information for each memory pattern generator gets combined at the formatter to achieve 200 MHz operation.

The IO channels must use channel-muxing on each channel that is to operate at 200 MHz. Drive-only channels can use either formatter-muxing or channel-muxing to operate at 200 MHz.

The J994 and J993 are also capable of operating in dual synchronous mode, allowing them to achieve an operating speed of 120 MHz.

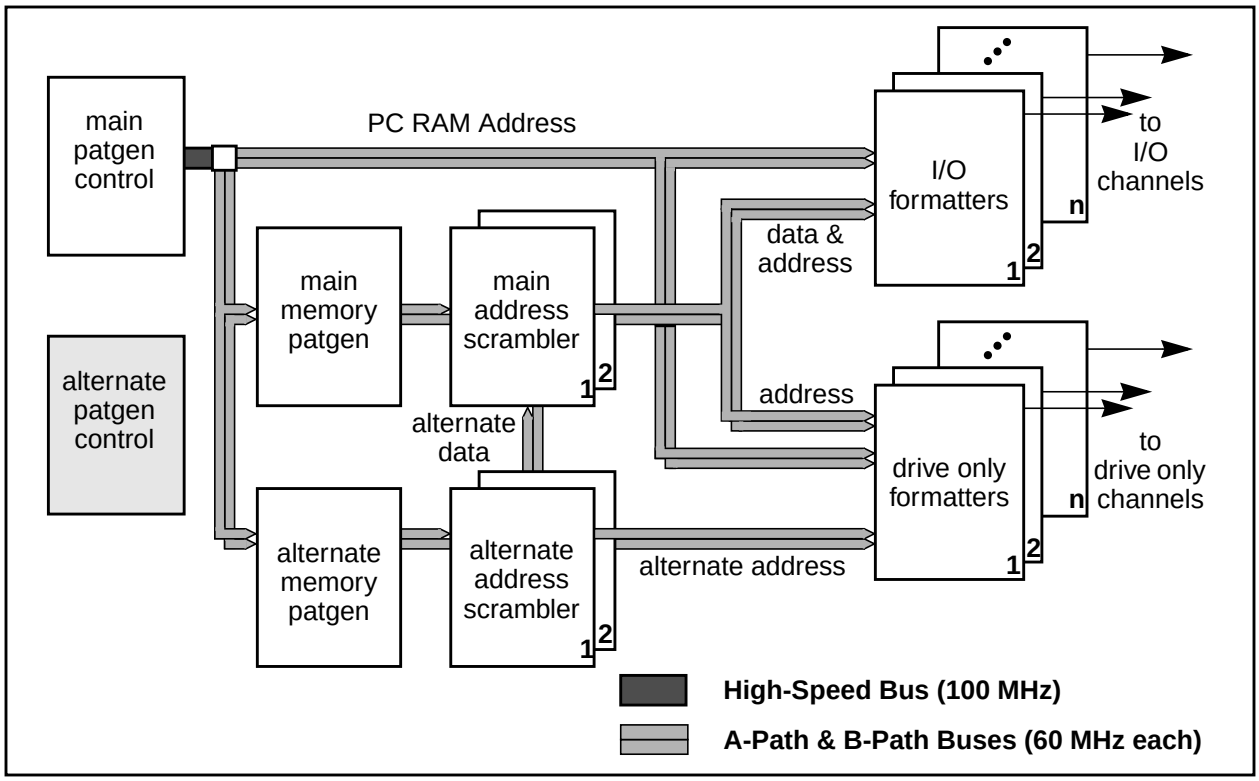


Figure 2.1-4: Dual Synchronous Patgen Operation (200 MHz mode)

Asynchronous Patgen Operation

The pattern generator controller which controls pattern flow and edgeset selection is separated from the memory pattern generator which generates algorithmic address and data. Two pattern generators and two memory patgens can be present in the system.

The two pattern generators are used independently to achieve asynchronous operation. This is useful for testing multiport devices such as video RAMs. Flag signals can be used for handshaking between the two patgens, so they can sync up when desired in the functional test pattern. The signals from each patgen drive different channels in the test head. The channels to be driven by each patgen are selectable.

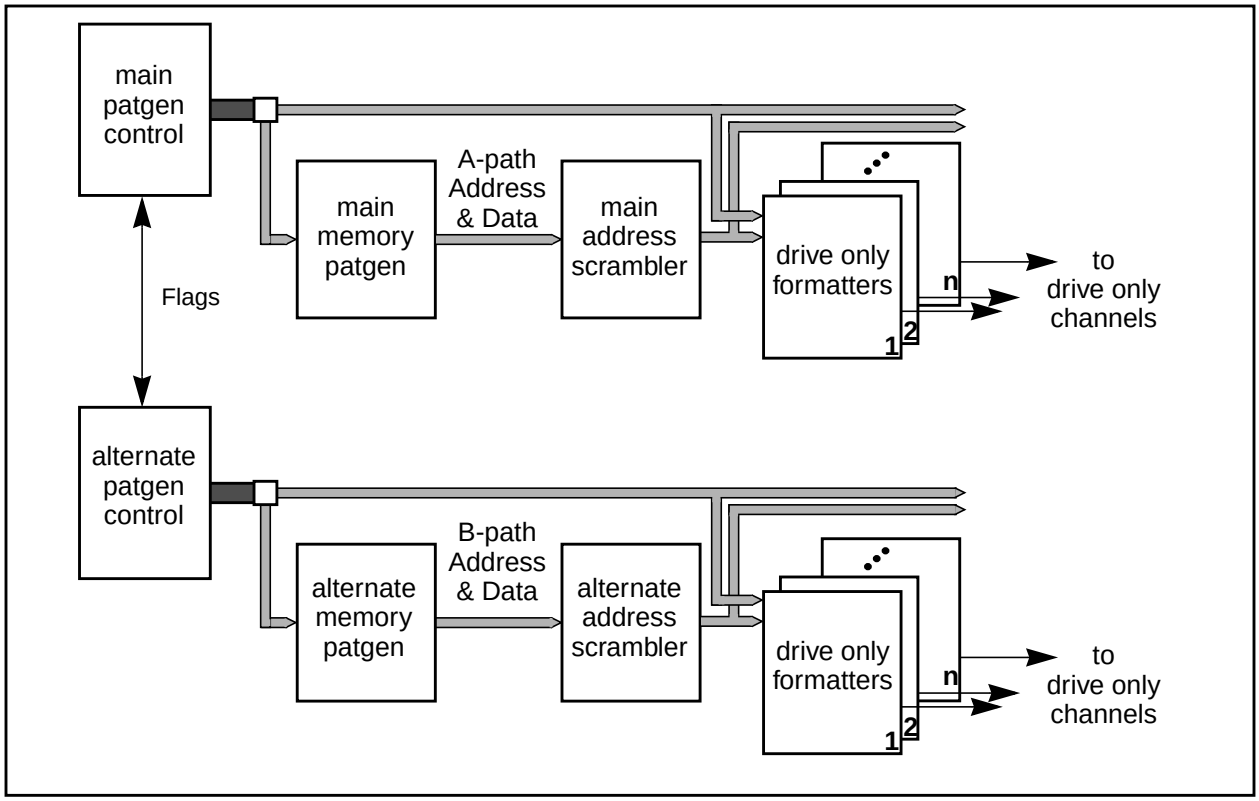


Figure 2.1-5: Dual Asynchronous Patgen Operation

Algorithmic Address and Data on the ADS bus

The patgen controllers drive the memory patgens, which generate algorithmic address and data for the memory device. The algorithmic address information (physical or memory array address) is routed through the address scrambler, which can generate device pin addresses (scrambled addresses). The data generator can generate algorithmic data information, which can be topologically inverted based on address bits.

The scrambled address information and data are put onto the Address-Data Source (ADS) bus, which routes this information to the waveform generators. Each pin's waveform generator can select a unique bit from this ADS bus to be used as a source of data. Up to 32 bits of address and data can be selected to be presented on the ADS bus.

In addition to the ADS bus another bus, the SEC (secondary) bus, exists. The SEC bus is a secondary source of algorithmic address and data for the DUT. Like the ADS bus it is 32-bits wide. The alternate memory pattern generator communicates with other hardware in the system through the SEC bus.

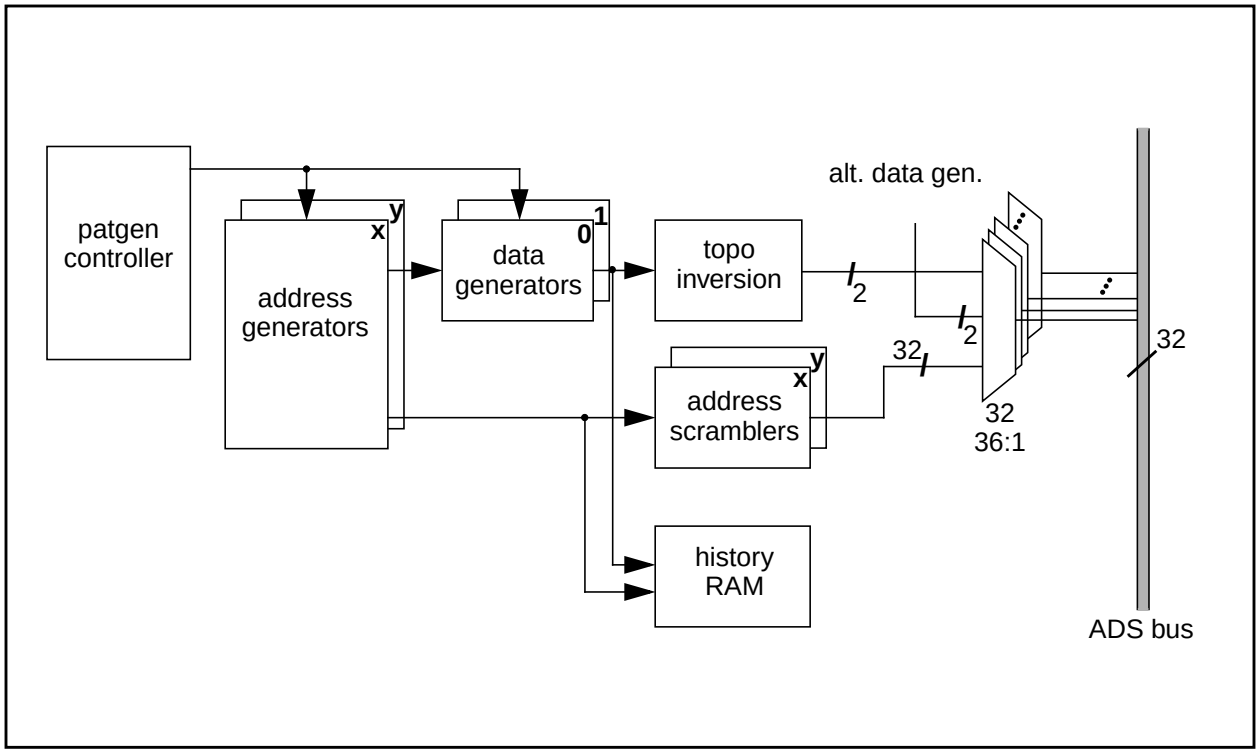


Figure 2.1-6: Address and Data Generation to the ADS bus

This address and data information is also sent to the History RAM (also known as Fail Vector Memory or FVM). The FVM can store (in sequence) the address and data values that are generated by the memory patgen. The FVM also stores fail information from the IO channels, and is used when debugging a test program or characterizing device failures.

Pattern Control using the Subroutine Vector Memory (SVM)

There are 1024 locations in the Subroutine Vector Memory (SVM). Within each pattern control location there are values for:

- ❒ Opcode and operand — for pattern flow and control
- ❒ Memory patgen address counter controls and presets
- ❒ Address counter selections
- ❒ Arithmetic Logic Unit (ALU) control functions
- ❒ Selection of a location of the pin data source select RAM
- ❒ Data generator set and group selections
- ❒ Edgeset address location selection
- ❒ Error processing control bits — for Catch RAM and display processor
- ❒ Channel data (pin state) for each pin

The pattern controller operates at 100 MHz, with unrestricted operation

The SVM is distributed throughout the system. Some SVM hardware is contained on the pattern generators, some on the memory patgens, and some is present on the waveform generators.

The pattern controller also selects which device voltage source level (either main or alternate), will drive the device. This is used for VS bump testing.

The channel data controls the pin state on a cycle-by-cycle basis. For IO channels, this pin state data includes drive and compare modes. For drive-only channels, this channel data consists of logic state data for the driver. The source selects will determine how this pin state data is used.

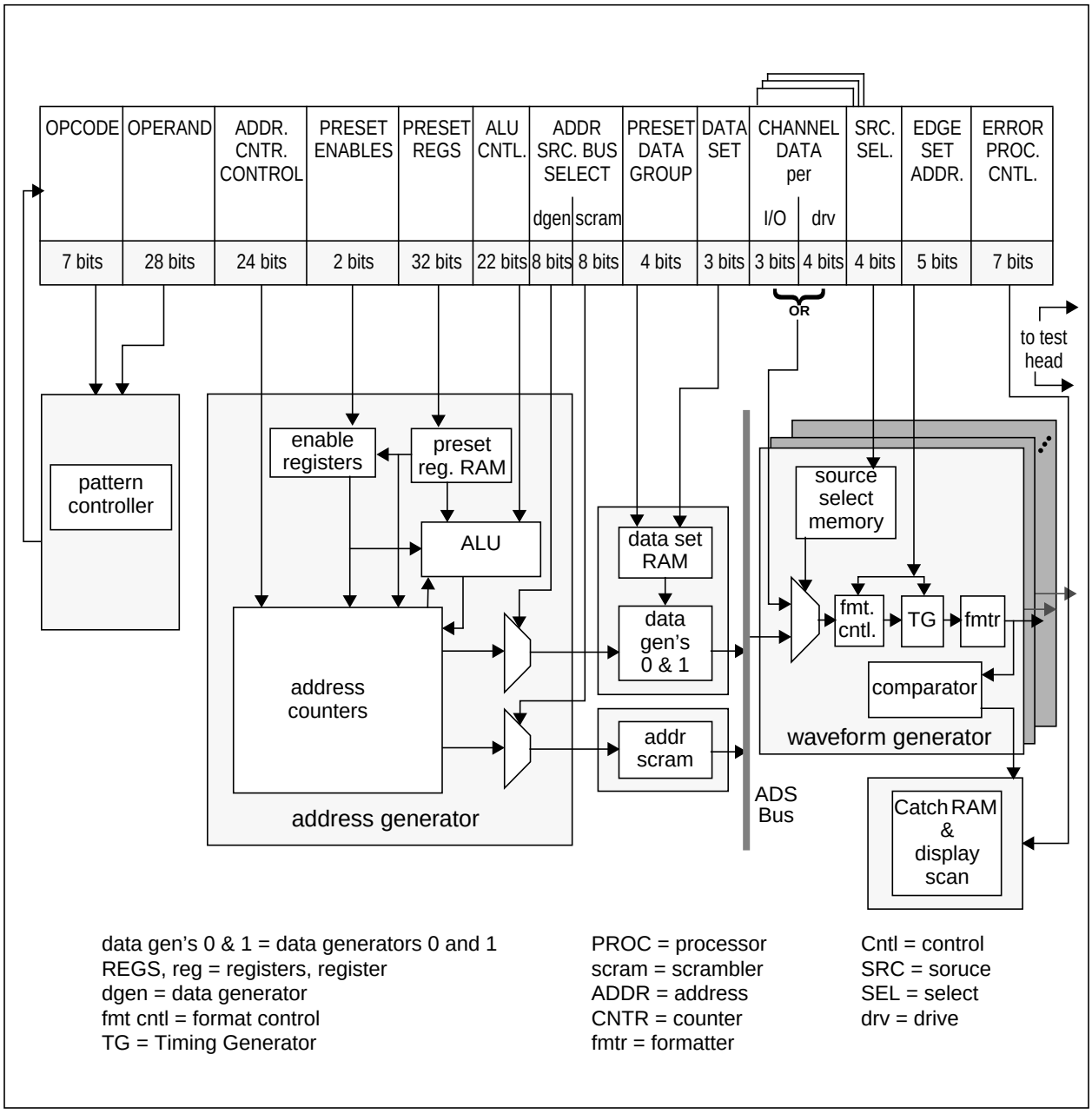


Figure 2.1-7: SVM Location Contents Diagram

Pattern Flow Control

To control the flow of the pattern, there are several counters which are used to keep track of the number of cycles (in a loop or on a single pattern location). These counters are set up with a value, then decremented until zero is reached.

- ❒ 28-bit repeat counter (using accumulator) to repeat a certain number of cycles on a single pattern step
- ❒ 4 nestable loop counters (28 bits) to program loops of multiple steps
- ❒ 7 utility counters which are independent of loop counters and each other, used for conditional branching of pattern flow
  - 4 counters have 28 bits (c1, c2, c3, and c4)
  - 3 counters have 14 bits (c5, c6, and c7)
- ❒ Asynchronous timed interrupt, with automatic branching to pattern subroutine for DRAM device refresh capability. Refresh interrupts are generated by an asynchronous 24-bit counter with 32.768 nS resolution. Refresh interrupts can be enabled, disabled, or forced.

The pattern controller also permits the calling and return from pattern subroutines, reducing the number of pattern locations needed. Pattern sequences can be shared between patterns by putting them into subroutines, and calling them from the main pattern flow. Pattern subroutines can be called from within a subroutine, since the pattern controller allows up to four nested calls.

Four user flags are also available to the pattern controller to affect pattern flow. These flags can be set, cleared, and compared for logic high and logic low, both in the pattern and by the CPU. These can be used to perform *handshaking* between the pattern generator and the CPU.

Address Generation

The memory patgens generate address values that drive the address scrambler and data generator. Each memory patgen generates a 16X, 16Y field. The address values for the data generator can be different from those used by the address scrambler.

X address values can come from one of five sources:

- ❒ XA, XB, XC, XD counters, and the X register in SVM

Y address values can come from one of five sources:

- ❒ YA, YB, YC, YD counters, and the Y register in SVM

The selected address source can also be inverted (complemented).

Each of the address counters (XA, XB, XC, XD, YA, YB, YC, YD) can be:

- ❒ incremented
- ❒ incremented-linked
- ❒ decremented
- ❒ decremented-linked
- ❒ loaded from SVM registers (masked by enable bits)
- ❒ jammed from SVM registers (not masked by enable bits)
- ❒ loaded from ALU (masked by enable bits)
- ❒ held (no action)

Address counter actions (increment, decrement, etc.) happen prior to the device cycle. This means that counter values should be preset to be one less than their desired starting value when the pattern step is incrementing the counters. They should be present to be one more than the desired starting value when they will be decremented.

The address enable fields prevent bits in the counters from being incremented, and control the effective address field. There are separate X and Y enable values, which can be preset on-the-fly, by loading from the SVM registers.

Address counters *cannot* be compared against registers in order to determine coincidence conditions for pattern flow control and branching. Flow control and branching are determined by the user flags, repeat, loop, and utility counters only.

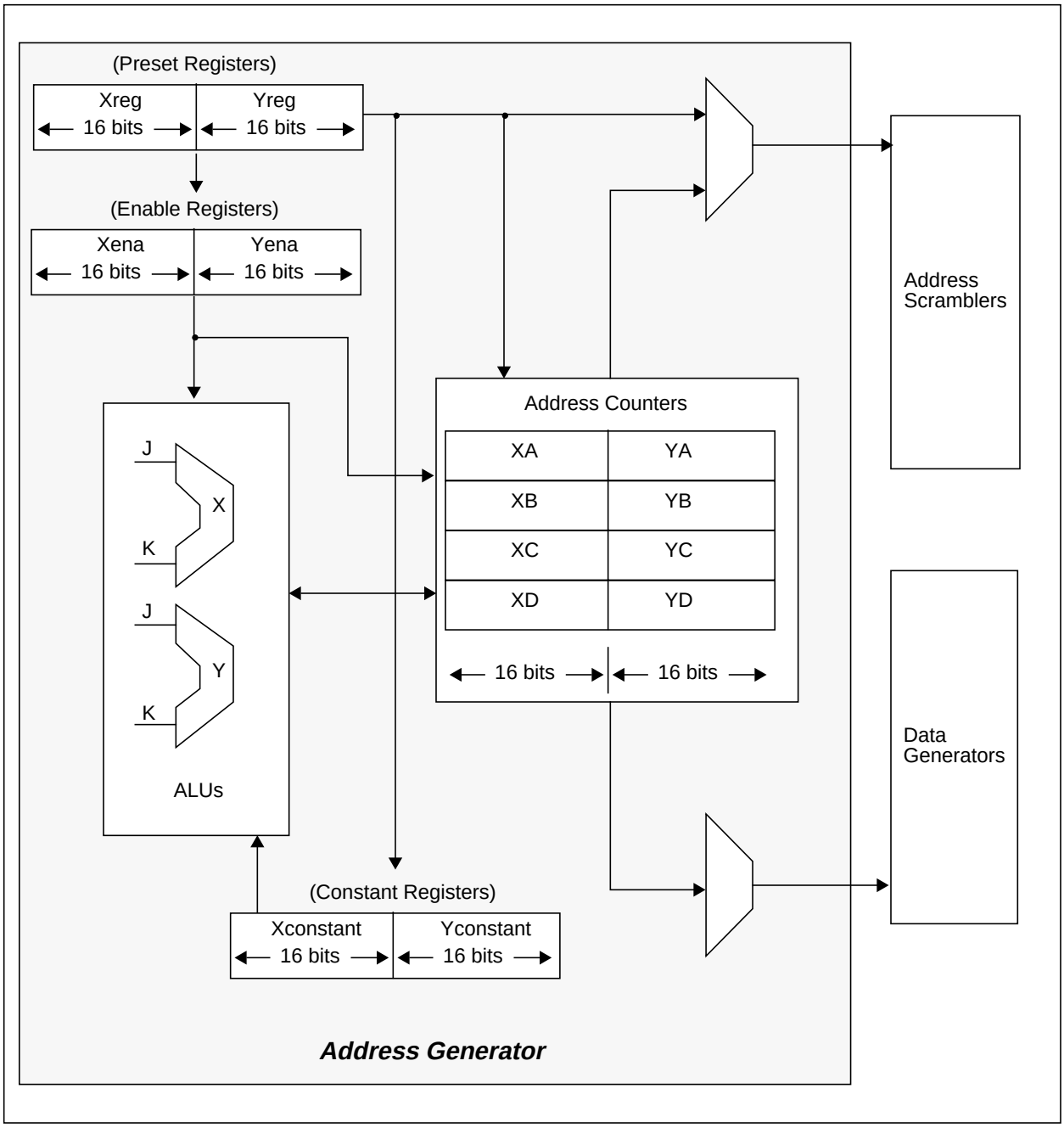


Figure 2.1-8: Address Generator (Mempat) Block Diagram

Arithmetic Logic Unit (ALU) for Address Generation

There are separate X and Y ALUs used to generate complex address sequences. The ALUs generate a value which can be loaded into the address counters and used as the source of address.

Each ALU has two inputs (J and K), which can be selected on-the-fly from one of:

- ❒ any counter (A, B, C, D)
- ❒ ALU constant value
- ❒ logic zero

One of six actions can be performed by each ALU. This action is performed on the current value of the J and K inputs.

- ❒ add
- ❒ add-linked
- ❒ subtract
- ❒ subtract-linked
- ❒ exclusive OR-linked (plus carry)
- ❒ exclusive NOR-linked (plus carry) — (complemented exclusive OR)

The ALU operations are masked by the enable bits.

The ALU constant can be loaded from the SVM registers or rotated on-the-fly. This becomes effective in the next cycle. When loading the ALU constant, all bits are updated regardless of the state of the counter enable register. When rotating the ALU constant, disabled bits are skipped. The ALU constant cannot load and rotate at the same time and rotation cannot be linked (X and Y will rotate independently).

Please note, there are two independent ALUs, one for X and one for Y. In the XOR and XNOR functions, carry-in and carry-out operate as follows:

> **Carry-out**: None — There is no carry-in to the other ALU when in XOR or XNOR mode.
>
> **Carry-in**: NOT ignored — I*f there were a carry-out* from the other ALU, it would affect the outcome.

Therefore, if both X and Y ALUs are in the XOR and/or XNOR modes, they would operate as expected -- with no interaction. However, a carry would be generated if one of the ALUs is in XOR or XNOR mode while the other is in an add, add-link, subtract, or subtract-link mode. In short, a carry from an ALU can only occur when the ALU is doing an add, add-link, subtract, or subtract-link. Therefore, to avoid having an ALU produce an undesireable carry that could affect an XOR or XNOR operation on the other ALU, make sure that **both** ALUs are doing XOR or XNOR operations.

The ALUs will compute their results according to the function (action) programmed in the current cycle. It is important to note that the values of the ALU inputs will come from the previous cycle. Also, the ALU constant must be preset one cycle prior to its use.

Address Scrambling

The address source selected by the memory patgen is used as inputs to the address scrambler. The address scrambler generates device address values based on the algorithmic address values from the patgen, by accessing a location in the scramble RAM.

The address scrambler is divided into an X half and a Y half, but since both X and Y address information can be affected, this separation is for naming purposes only. The X scrambler can affect up to 14 bits of address (16K locations), and the Y scrambler can affect up to 16 bits of address (64K locations). Both of these can be combined to generate up to 30 bits of scrambled address for the ADS bus. In addition, two address bits are unscrambled and can be selected to go directly to the ADS bus.

The bits to be scrambled are selected at the input to the scramble RAM, and outputs (scrambled address) can be selected to be routed to the ADS bus.

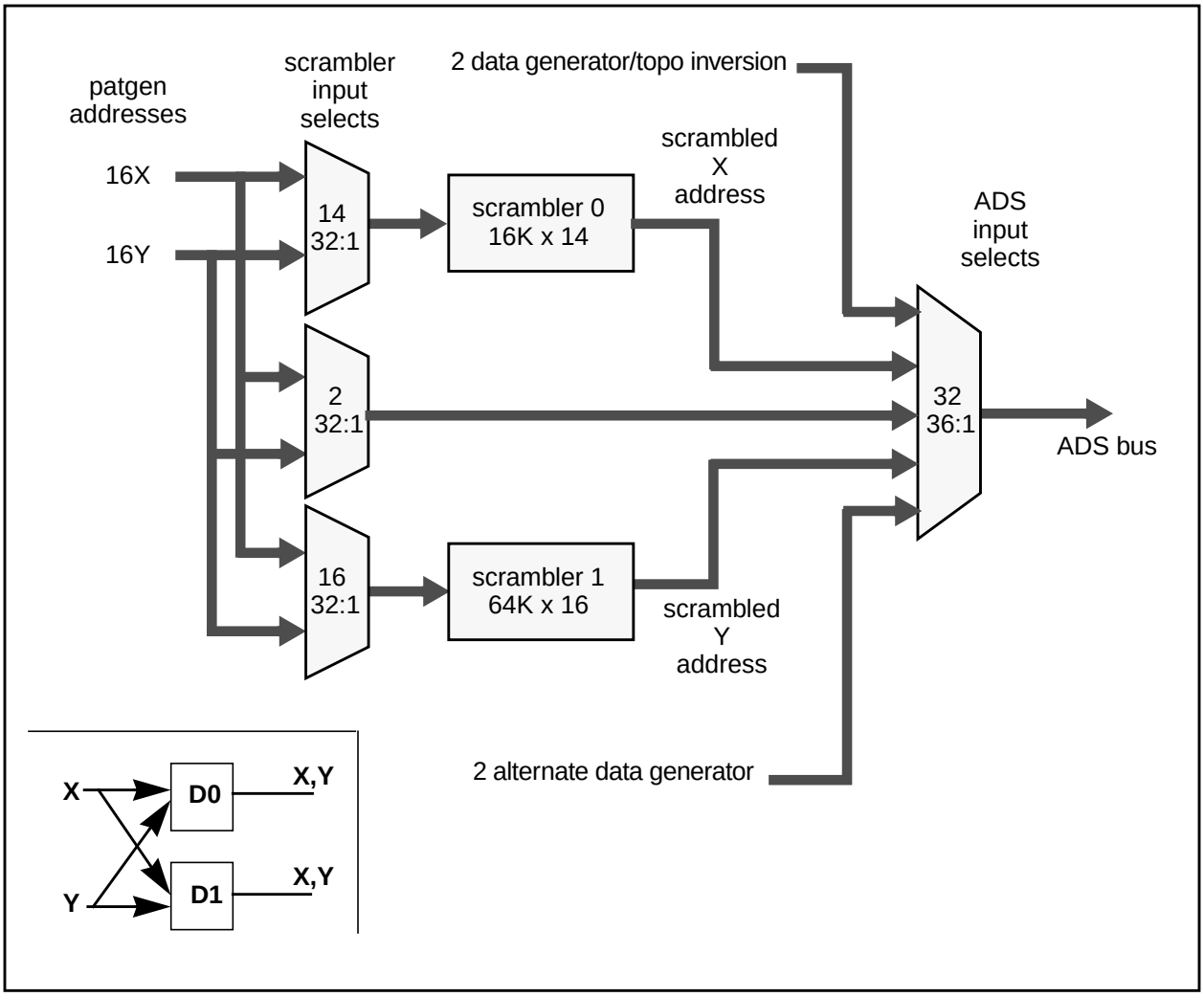


Figure 2.1-9: Address Scrambler Block Diagram

Data Generation

Each memory patgen has two data generators, each generating one bit of algorithmic data for the device. Data can be from the:

- ❒ Logic Function Generators (LFGS), which can generate data as a function of address
- ❒ Coincidence (diagonal) data, when row (X) address equals the column (Y) address data will equal 1
- ❒ Logic 0

Each data generator selects one of these data sources, which can be inverted. For example, logic 1 data can be generated by inverting the logic 0 selection. Selection of these functions is available on-the-fly (in the data set).

In addition, the function of each LFG is programmed on-the-fly (in the data set). The function of the LFG can be AND, OR, NOR, XOR, and so on (see page 2.1.16). These functions are performed on the two address bits that are selected to drive each LFG. The address selections are done with the databus, prior to the patgen burst.

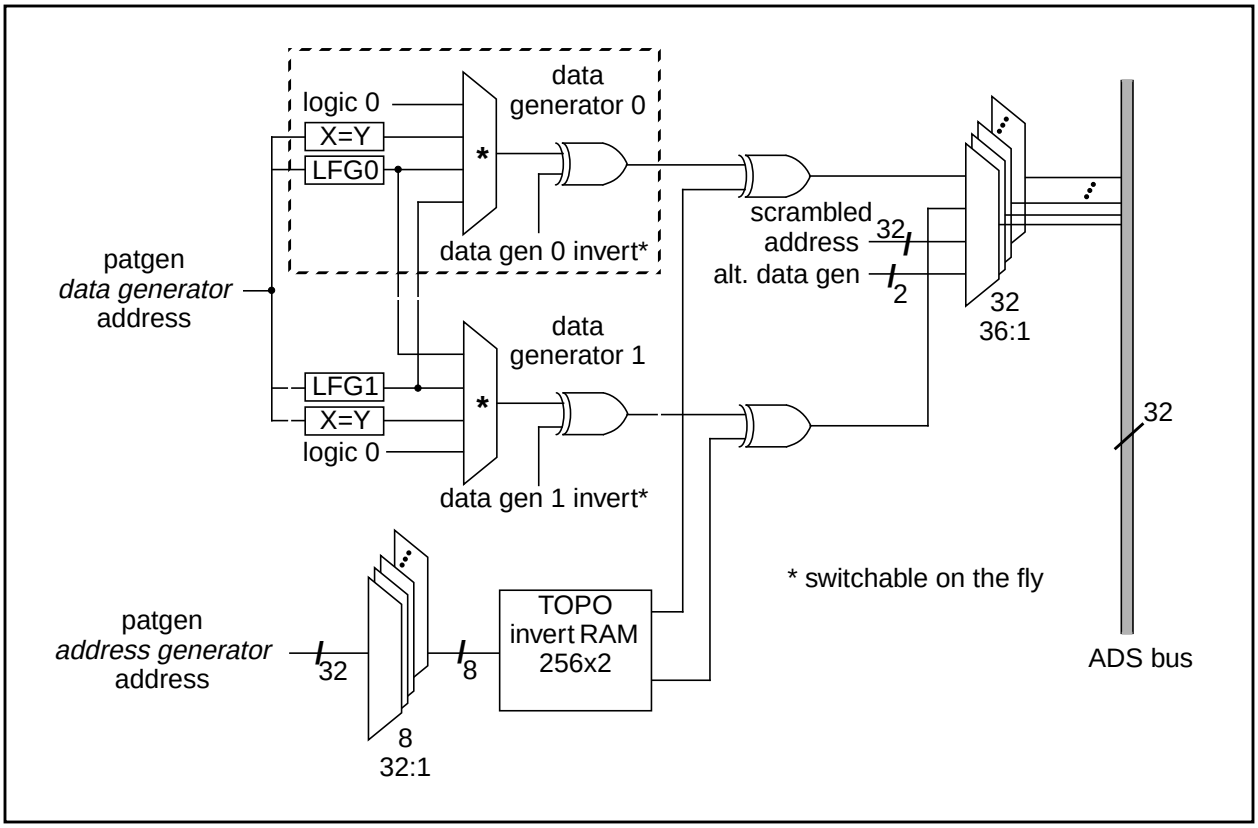


Figure 2.1-10: Data Generation on the Memory Patgen

This data can also be inverted based on selectable address bits (topological inversion), so data is generated based on how the device array will store the data. Because the data generation is two bits wide, two bits of topo inversion are available for each topo RAM. The topo RAM is implemented as four separate 256kx2 blocks.

The two data generator data bits are available for selection onto the ADS bus, making these bits available to each channel's waveform generator.

Table 2.1-1 LFG Functions

LFG Output	
logic 0	logic 1
a	b
~a	~b
a OR b	~a OR b
a OR ~b	~a OR ~b
a AND b	~a AND b
a AND ~b	~a AND ~b
a XOR b	a XNOR b

Data is programmed in eight groups of eight sets each. Data sets are selected by each pattern instructions. Data groups can be changed at any point in the pattern, and take effect in the next cycle.

Waveform Generation

Waveform generation on J990 Series test systems is done uniquely for each channel. Each channel has 32 edgesets, which contain both format and timing information for a single cycle. The edgesets are programmed prior to starting the patgen. While the patgen is running, any of the 32 edgesets can be randomly selected to produce waveforms at the device.

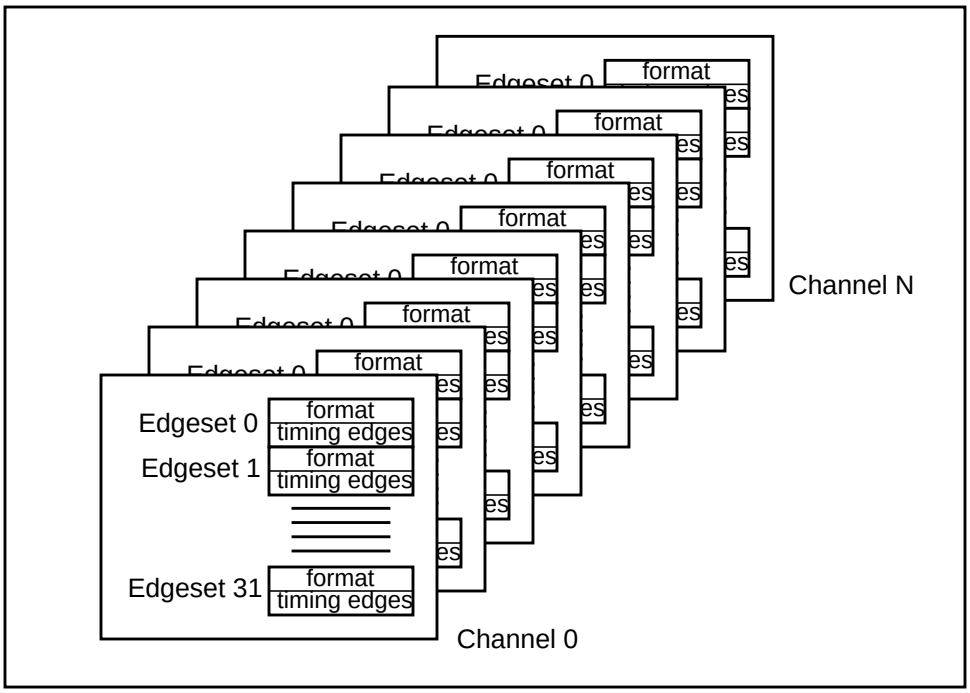


Figure 2.1-11: 32 Edgesets for Each Channel

Edgesets are typically programmed using waveform databases (waveadr or wavedb files). Both the timing edges and format are specified in the eqset statement in the waveform database.

Available Timing Edges

Each edgeset contains six timing edges for the channel. Each channel type (drive-only and IO) have a different set of edges as described below.

Drive-Only Channel Edges

Table 2.1-2
Drive Only Edges

Edge	Description
d0	Used to start the *surround data* for a cycle. The surround data depends on the channel's format.
d1	Starts (gates) the data for a cycle.
d2	Ends the data for a cycle, starting the surround data.[a]
d3	Format mux edge used to switch to the second piece of data for a cycle. Commonly used to switch to column address when address-multiplexing.
d4	Starts (gates) the second piece of data for a cycle.
d5	Ends the second piece of data for a cycle, and starts the surround data

a. The d2 edge ends the data gate except when using Mach2 mode, in which d2 starts the second piece of data from the SVM.

The enabled modes for each drive-only channel determines how many of the six available edges will be used in a cycle. For example, edges d3, d4, and d5 are only used when format-muxing the channel.

IO Channel Edges

Table 2.1-3
IO Edges

Edge	Description
d0	Used to start the *surround data* or change the drive/HiZ state for a cycle. The surround data depends on the channel's format.
d1	Starts (gates) the data for a cycle.
d2	Ends the data for a cycle, starting the surround data.
d3	Used to end the surround data or change the drive/HiZ state for a cycle.
c1	First strobe edge of comparator. Used to start the window strobe for the channel.
c2	Second strobe edge of the comparator, used to end the window strobe interval. Used as the only edge in edge strobe mode

The enabled modes for each IO channel determines how many of the six available edges will be used in a cycle, and how they will be used. For example, edges c1 and c2 are only used when expecting device output data and the driver is HiZ. Edges d0 and/or d3 can be used to change the drive/HiZ state of the channel.

When in strobe window mode, c1 is programmed to occur before c2. When in strobe edge mode, c1 should be programmed to occur at the same time as c2.

Driver Formats

On a cycle by cycle basis, the format can be changed for each channel by selecting a new edgeset. The data shown in each figure below can come from the pattern generator or SVM channel data, depending on the programming of the source selects. The formats to choose from include:

Return to Low (RL): Logic low from d0 to d1, valid data from d1 to d2 (gate edges), and logic low from d2 until d3 (or next cycle's d0)

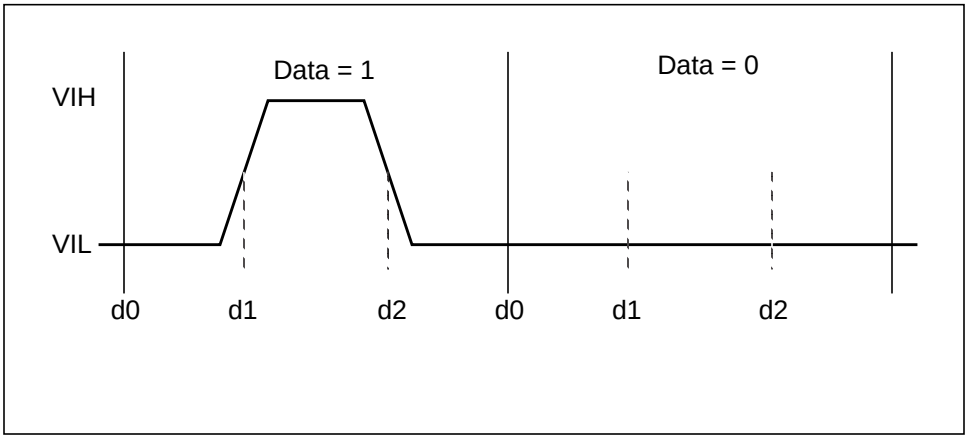


Figure 2.1-12: Return to Low Format

Return to High (RH): Logic high from d0 to d1, valid data from d1 to d2 (gate edges), and logic high from d2 until d3 (or next cycle's d0).

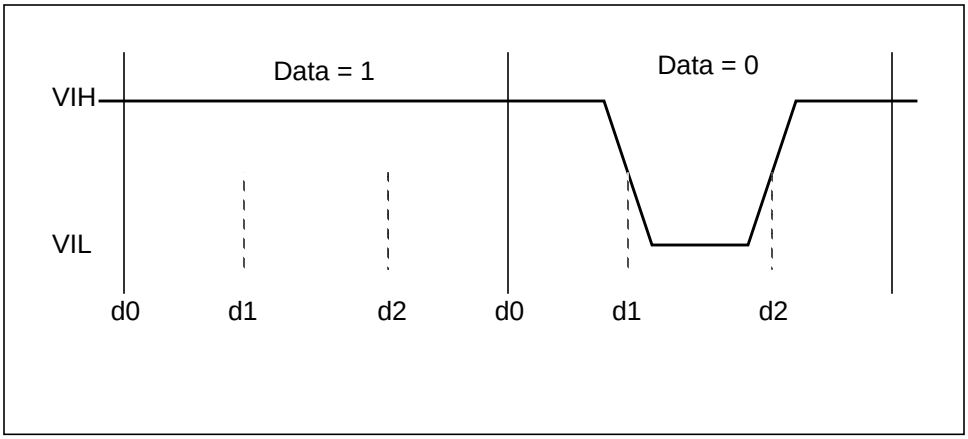


Figure 2.1-13: Return to High Format

Pulse Low (PL): Regardless of the actual data, the channel goes high from d0 to d1, low from d1 to d2, and high from d2 to d3 (or the next cycle's d0).

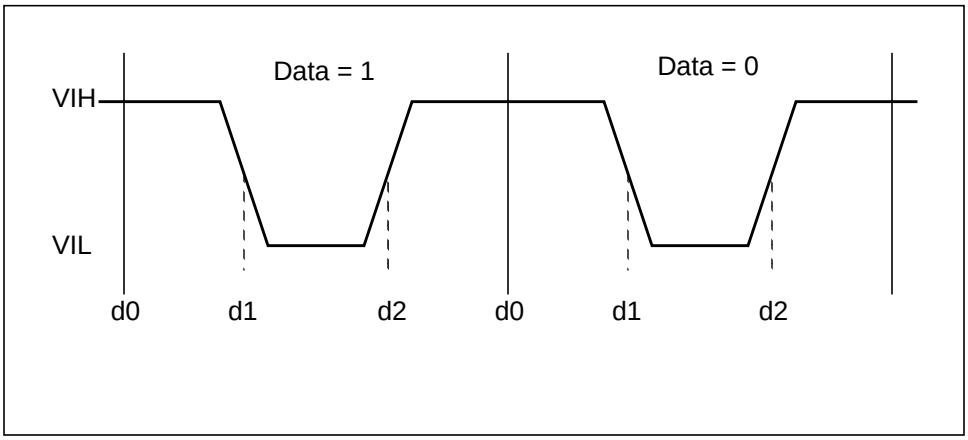


Figure 2.1-14: Pulse Low Format

Pulse High (PH): Regardless of the actual data, the channel goes low from d0 to d1, high from d1 to d2, and low from d2 to d3 (or the next cycle's d0).

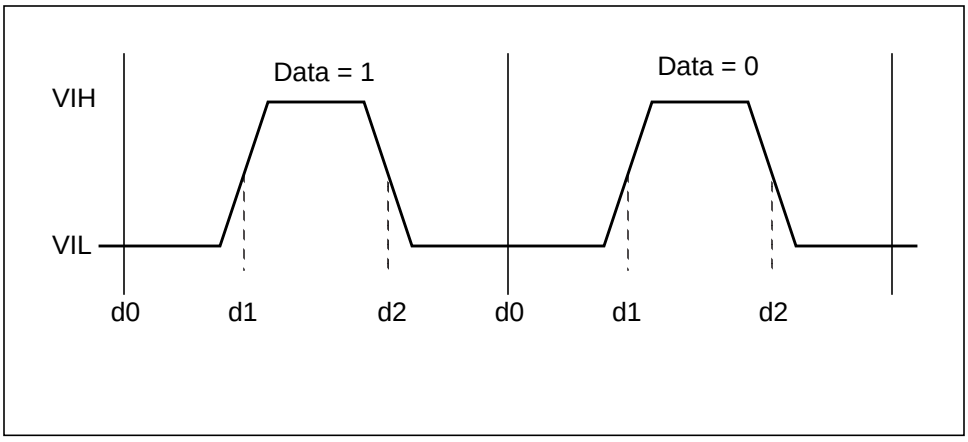


Figure 2.1-15: Pulse High Format

Return to Complement (RC): Complement data from d0 to d1, data from d1 to d2, returning to complement data from d2 to d3 (or the next cycle's d0).

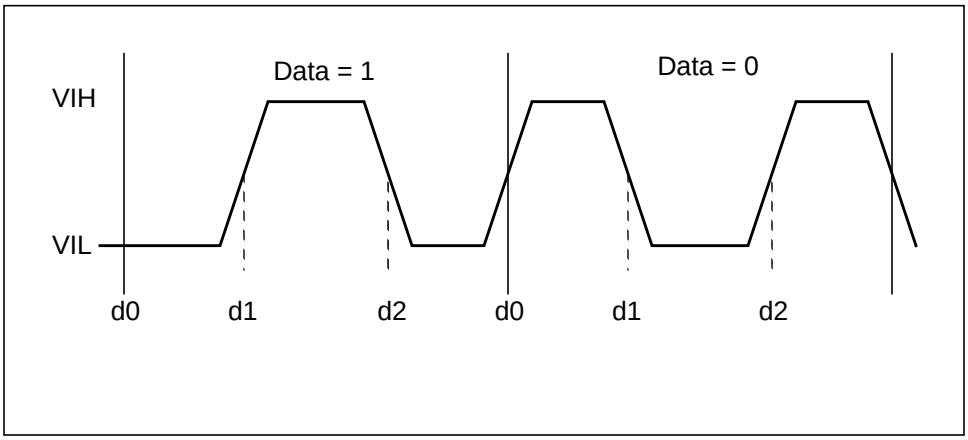


Figure 2.1-16: Return to Complement Format

Non-Return (NR): Data from d0 to d1 is determined by previous cycle, valid data starts at d1, continuing until next cycle's d1 (or possibly next cycle's d0, depending on the next cycle's format). Edges d0, d2, and d3 are effectively ignored when using this format.

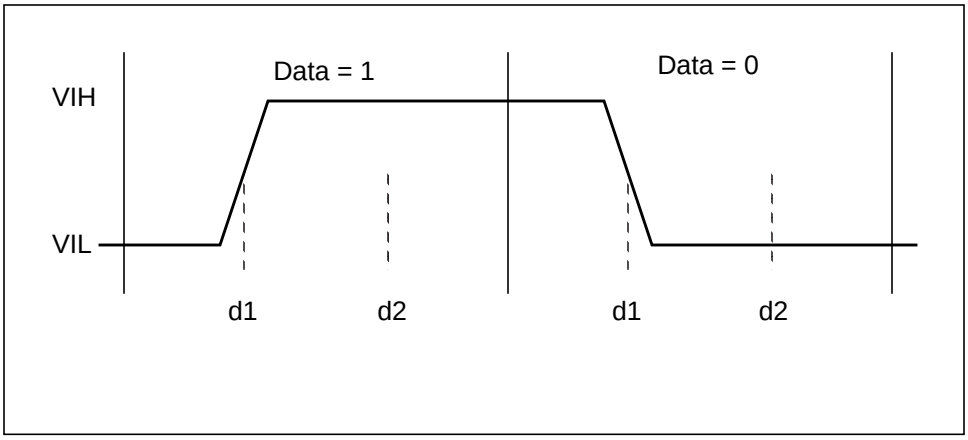


Figure 2.1-17: Non-Return Format

Stay (ST): Regardless of the actual data, keep the logic state that was present at the end of the previous cycle. All timing edges are effectively ignored during the cycle.

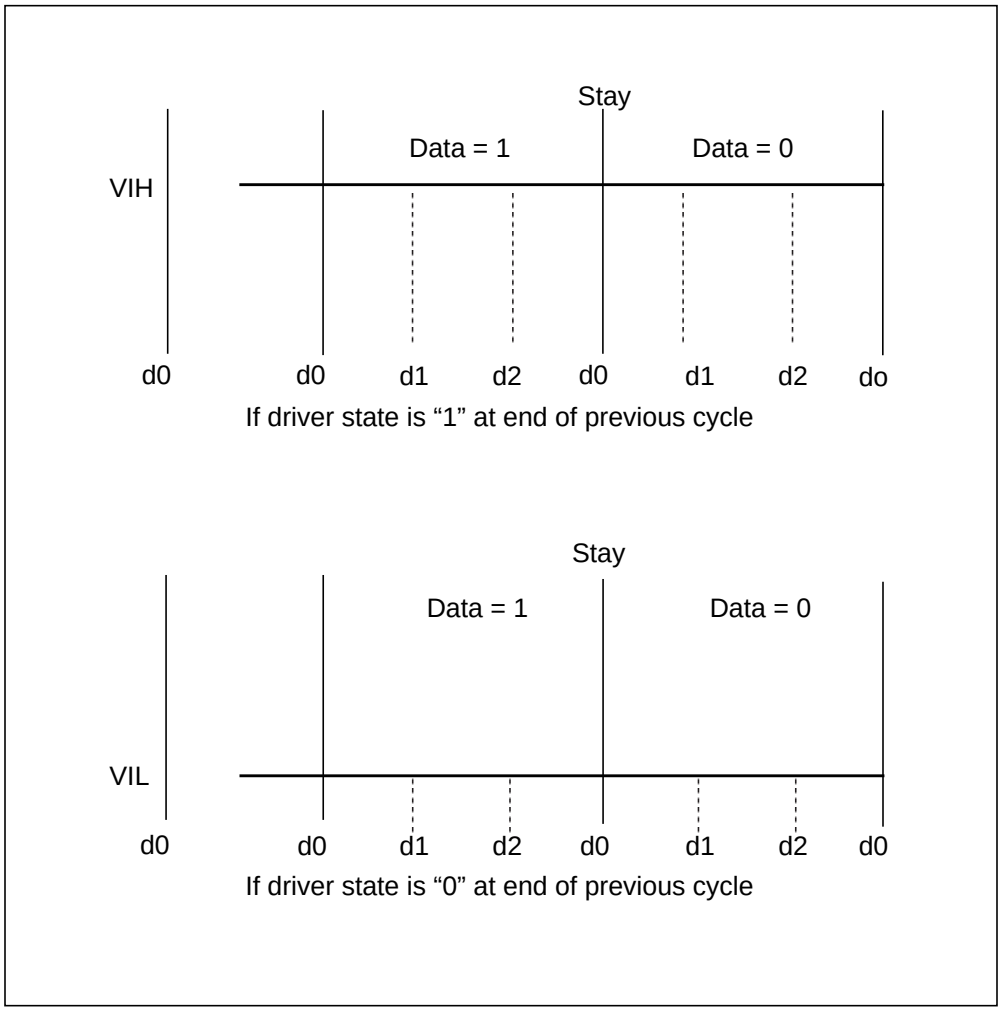


Figure 2.1-18: Stay Format

Copy (CP): Copy (or repeat) the format and data that was used in the previous cycle.

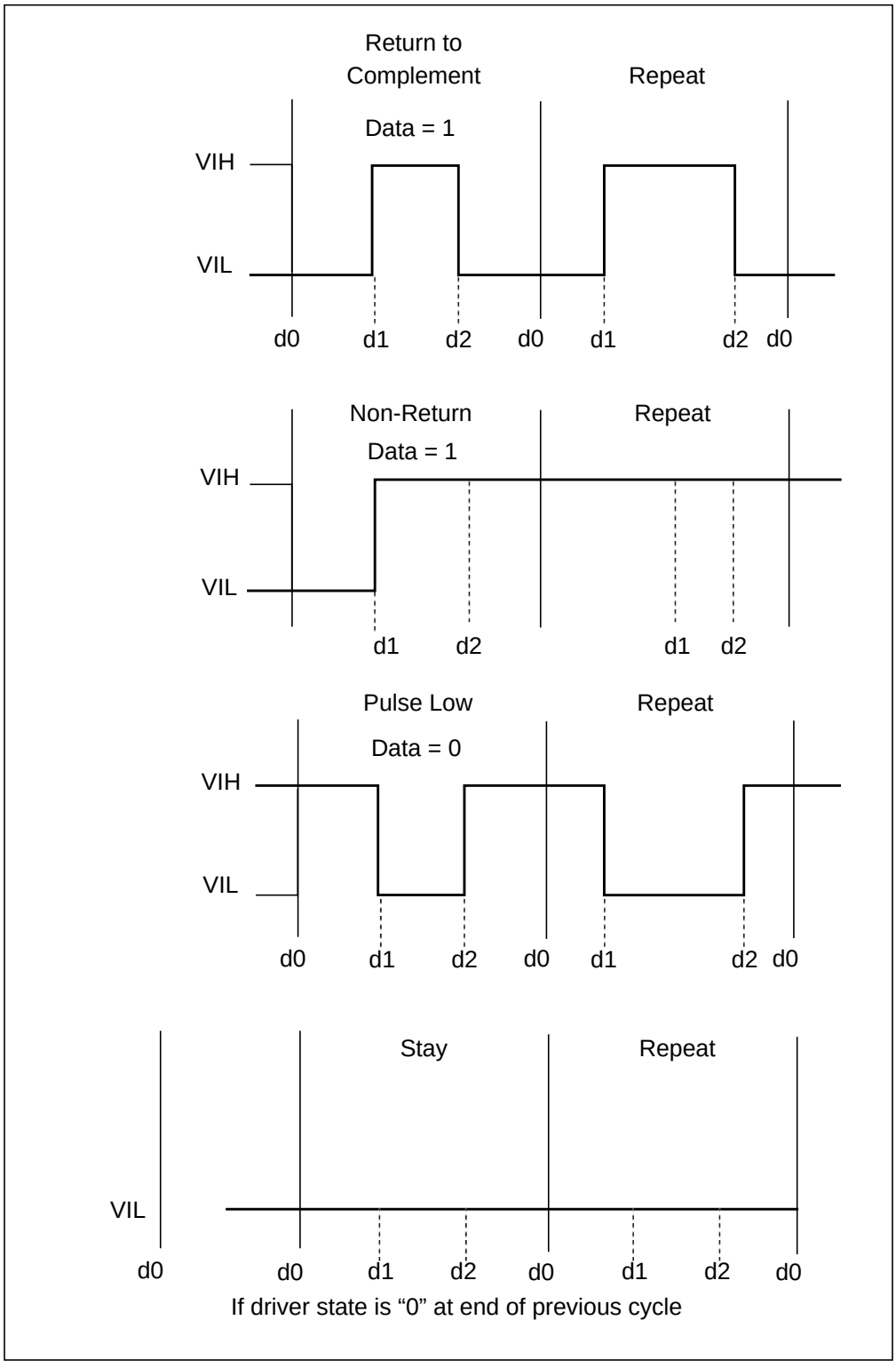


Figure 2.1-19: Copy Format

Format Muxing on Drive-Only Channels

Format-Muxing (on drive-only channels only) allows the channel to drive two independent pieces of data in one cycle. This is commonly used to generate address multiplex waveforms, and clock the channel at twice the patgen speed. The driver format is independent in each half of the cycle.

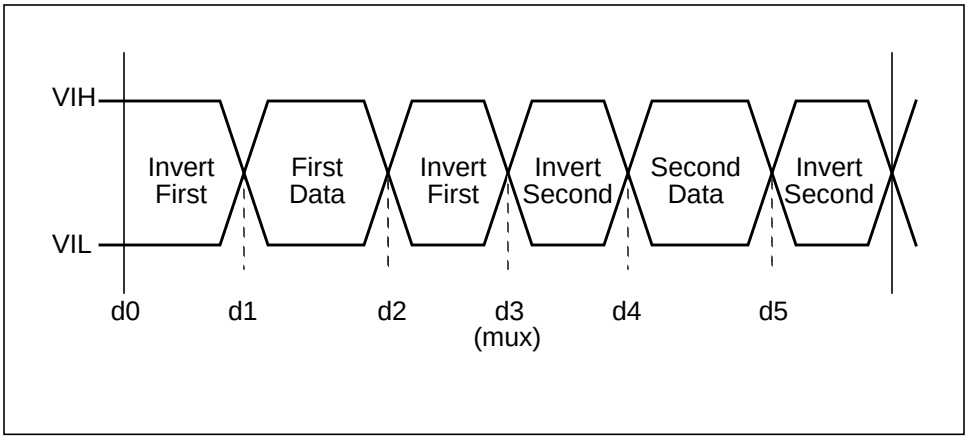


Figure 2.1-20: Formatter Muxing with RC, RC

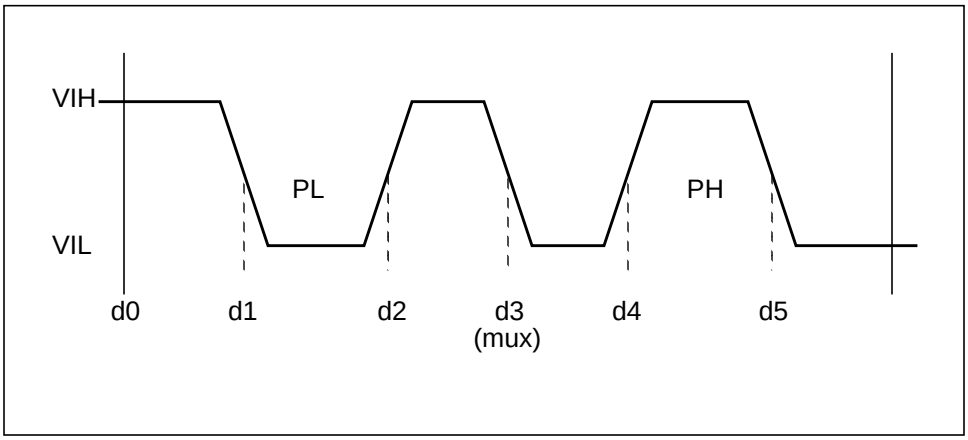


Figure 2.1-21: Formatter Muxing with PL, PH

Channel Muxing

Channel muxing utilizes two channel's edges on a single pin to generate waveforms at twice the formatter speed. This muxes the adjacent channel's edges onto the current channel. All programmed edges for both channels are combined into one cycle. Channel muxing is possible on both the drive-only and IO channels. For drive-only channels, the d0 edge of the adjacent channel determines when that channel's edge will mux onto the current channel. For IO channels, the d3 edge of the adjacent channel determines when that channel's edges will mux onto the current channel. The d3 edge of the current channel (in the next cycle) toggles back.

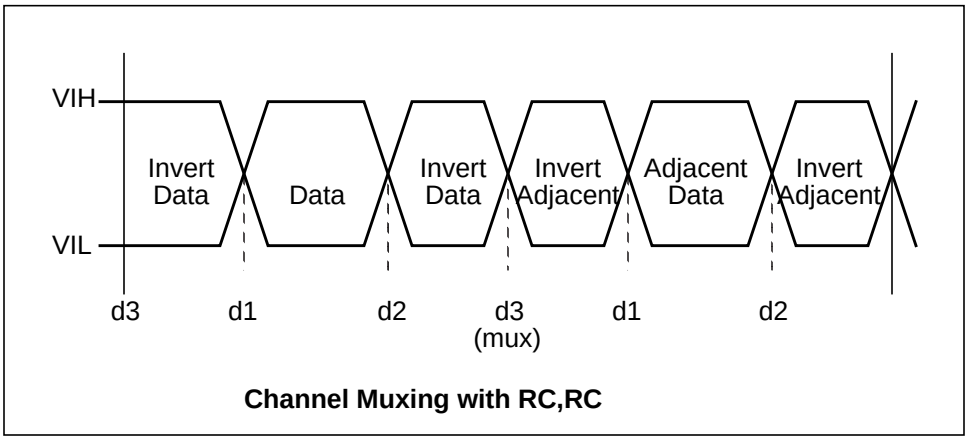


Figure 2.1-22: Channel Muxing with RC, RC (IO channels)

While very high frequencies can be achieved with this mux mode, the adjacent channel may not be usable.

It is possible to use formatter-muxing and channel muxing on the drive-only channels to get four independent pieces of drive data in one cycle. The only limitation would be the analog performance of the channel driver itself.

Channel muxing is also used to do double strobing in a cycle. Two adjacent IO channels are muxed together, which combines the drive and detect portions of both channels. Channel muxing on IO channels can be used to get an edge strobe and a window strobe in the same cycle.

An example of a set of timing that works for chan-mux under zstate d0 on an IO pin in inout mode.

```
(pins-form
                (DQS_even :io (zstate d0 )
                        :scio off :comp on :strobe edge
                        :chan-mux on)
                (DQS_odd :io (zstate d0 )
                        :scio off :comp on :strobe edge
                        :chan-mux on)


                (read_edges     :per "10ns"
;  When detecting, d0-d3 don't matter, just c1 & c2
                        (DQS_even    rc (d0     "0ns")
                                        (d1     "1.25ns")
                                        (d2     "3.75ns")
                                        (d3     "5ns")
                                        (c1     "DQS_even.c2")
                                        (c2     "3ns")
                        )
                        (DQS_odd   rc   (d0     "0ns")
                                        (d1     "1.25ns")
                                        (d2     "3.75ns")
                                        (d3     "5ns")
                                        (c1     "DQS_odd.c2")
                                        (c2     "8ns")
                        )
                (write_edges    :per "10ns"
;  When driving, d3 switches to that channel
                        (DQS_even    rc (d0     "0ns")
                                        (d1     "1.25ns")
                                        (d2     "3.75ns")
                                        (d3     "0ns")
                                        (c1     "DQS_even.c2")
                                        (c2     "6ns")
                        )
                        (DQS_odd   rc   (d0     "0ns")
                                        (d1     "6.25ns")
                                        (d2     "8.75ns")
                                        (d3     "5ns")
                                        (c1     "DQS_odd.c2")
                                        (c2     "6ns")
                        )
```

Mux Mode Programming

The mux modes (channel-mux and format-mux) are programmed in the **pins** statement in the waveform database. Whenever that waveform database is selected, the pins are programmed for their selected mode. The waveform database also contains the format and edge timing values for each edgeset for each channel being used.

Mach2 Mode

Each channel can also be put into a mode to receive two bits of logic state data from the patgen. This special mode is called Mach2, and it allows one patgen logic bit to be driven at d1, and a second bit to be driven at d2. These logic state bits can be programmed uniquely for each channel in the SVM.

This mode offers the most flexibility in generating logic signals for device clocks, since the programmer can independently control the channel driver states at d1 and d2.

Mach2 mode is only used with Non-return (NR) formats for the channel. Driver transitions will not occur at d0 and d3.

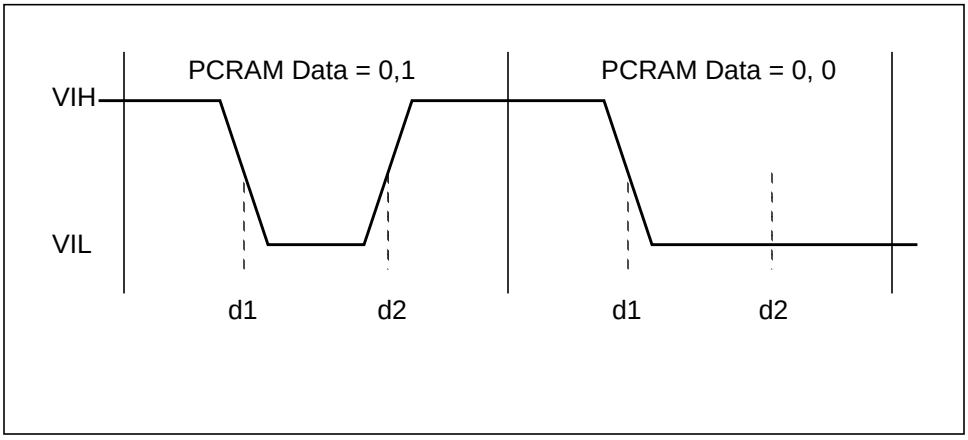


Figure 2.1-23: Using Mach2 Mode with Non Return Format

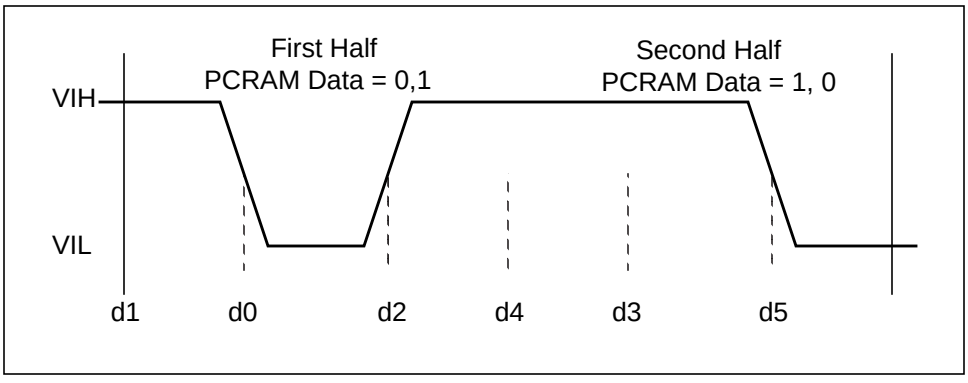


Figure 2.1-24: Using Mach2 Mode and Formatter Muxing with Non Return Format

IO Compare Modes

In addition to driving data, IO channels can also compare data. Either window strobes or edge strobes can be selected. Figure 2.1-25 shows examples of edge and window strobe. When programming edge strobes, the c2 edge is used to strobe. The c1 edge should be placed on top of the c2 edge (that is, c1 should be programmed to the same value).

Note also, as shown in the figure, edges can be programmed across the cycle boundary (d0). For edge strobes that means the strobe can be delayed into the next cycle, or several cycles. This is shown in the figure when the c2 edge strobe of the second cycle is delayed into the third. The normal c2 edge strobe for the third cycle also occurs as programmed.

For window strobing both c1 and c2 are used to define the length of the strobe. Like edge strobe, window strobes can be delayed into later cycles. Either the entire window strobe can be delayed into a later cycle (see cycles 5 and 7 in the window strobe example), or the c2 edge can be delayed across cycle boundaries resulting in a long window strobes that traverse multiple cycles. This is shown in the multiple cycle window strobe example.

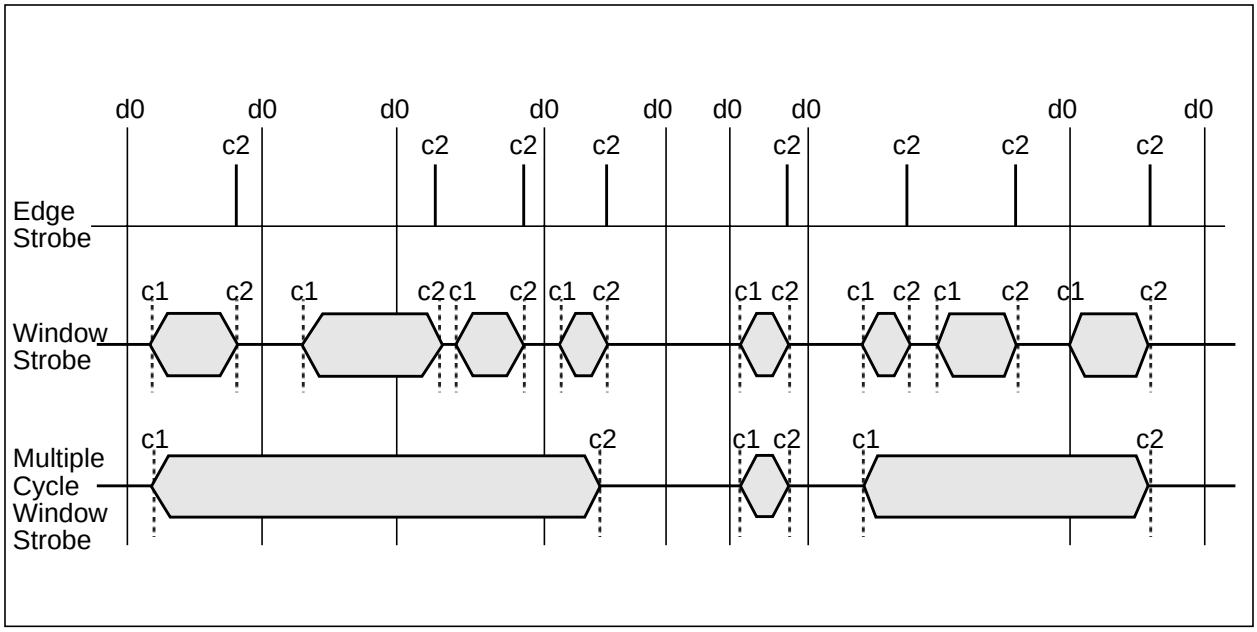


Figure 2.1-25 Examples of Edge and Window Strobes

Figure 2.1-26 illustrates the four types of window strobes you can program. The conditions are: expect high, expect low, expect midband, or expect valid. The window strobe conditions are determined with respect to the programmed values of VOH and VOL. These output voltages are the voltages used on the comparator circuit on the IO channel cards (see Figure 2.1-43 on page 2.1.51).

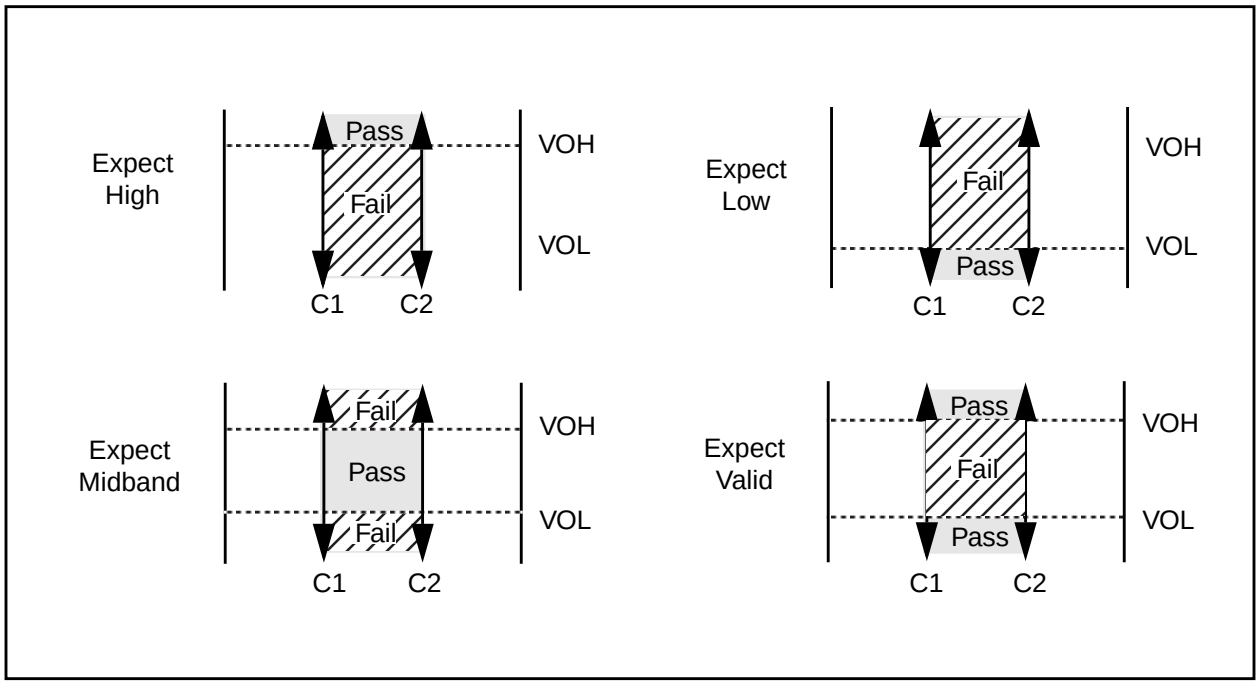


Figure 2.1-26 Window Strobe Formats

IO Switching Modes

The IO channels can also be set up to perform IO switching, which is necessary for testing common IO devices. As discussed earlier, the valid data will normally appear between the gating edges d1 and d2. Edges c1 and c2 control the strobe timing. The remaining edges d0 and d3 can be used to control the IO switching, which affect when the driver is on (LoZ) and when it is off (HiZ).

The IO channel can be put into one of four IO switching modes (also known as zstate selection), which is selected prior to running the patgen:

zstate d0: IO switch time at d0. For cycles with the comparator enabled, the driver will switch off (go to HiZ) at the d0 edge time. For cycles that are not comparing, the driver will switch on at d0.The d0 edge can be placed anywhere in the cycle.

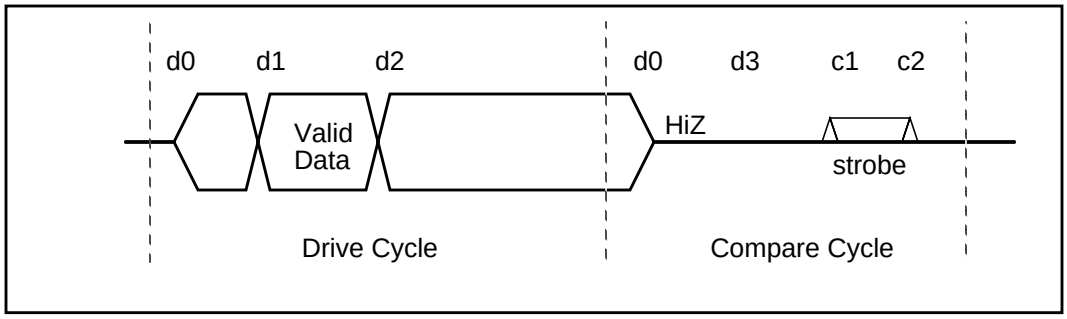


Figure 2.1-27: Zstate d0 Waveforms

zstate d3: IO switch time at d3. For cycles with the comparator enabled, the driver will switch off (go to HiZ) at the d3 edge time. For cycles that are not comparing, the driver will switch on at d3. The d3 edge can be placed anywhere in the cycle.

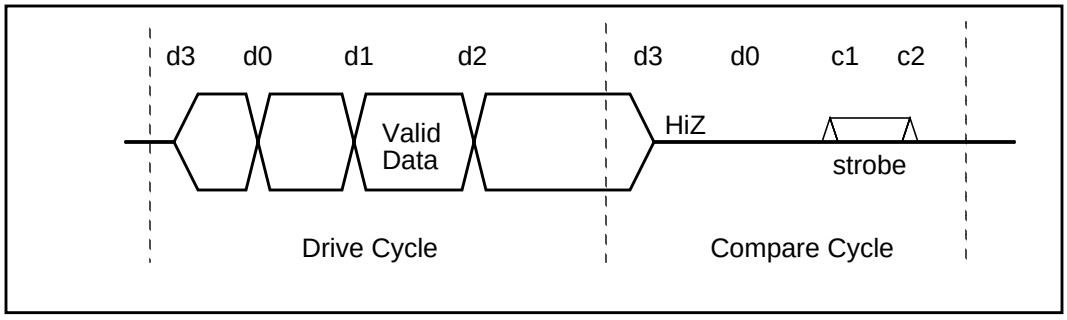


Figure 2.1-28: Zstate d3 Waveforms

zstate d0 off d3: IO switch time at d0 and d3. For cycles that are designed as "drive", the driver will switch on (drive) at d0 edge time and off (go to HiZ) at the d3 edge time. For cycles that are not driving, the driver will remain at HiZ.

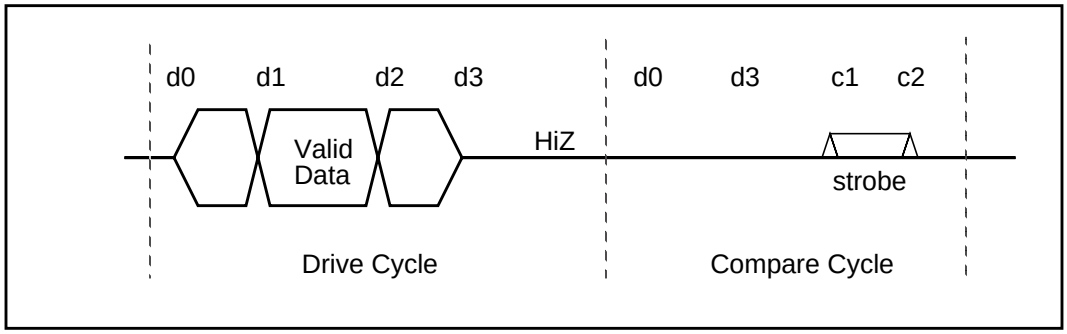


Figure 2.1-29: Zstate d0 off d3 Waveforms

zstate d3 off d2: IO switch time at d3 and d2. For cycles that are designed as "drive", the driver will switch on (drive) at d3 edge time and off (go to HiZ) at the d2 edge time. For cycles that are not driving, the driver will remain at HiZ

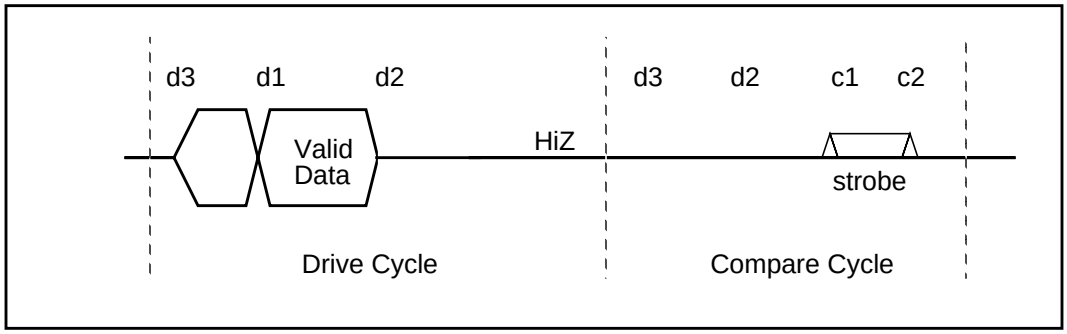


Figure 2.1-30: Zstate d3 off d2 Waveforms

These modes are selectable in the waveform database by using the :io clause in the pins statement.

Pin Data Sources and Source Select Memory

Introduction

In general, there are two sources of data for the channels: the Address-Data Source (ADS) bus, and the SVM.

The ADS bus is 32 bits wide and provides algorithmic data from the address and data generators. Each IO channel can select one or two of the 32 bits from this bus, which can provide both the drive and expect data for the channel. Drive-only channels can select one bit from the ADS bus.

The SVM can also provide data for the channels. The SVM pin data can be used to affect the driver logic state, expect data, drive/compare mode, and LoZ/HiZ state.

- ❒ Up to three bits of SVM data can be used for the IO channel to selectively decode to one of eight states.
- ❒ Up to four bits of SVM data can be used for the drive-only channel state.

Selecting the source of data (either ADS bus or SVM) is required for each channel being used. This is programmed in the source select database (srcadr/scrdb files) using the srcset statement.

The 8 State Model for IO Channels

The state of an IO channel in an individual cycle is determined by three bits. These three bits are decoded into an eight state model. The eight available states depends on the mode of the channel, but in general terms, the state of the channel is selectable on-the-fly.

For example, the table below illustrates the available eight states for the IO channel when operating in the normal mode:

Table 2.1-4
Eight State Model (Default Mode)

State	Description	Programming Symbol
0	Drive Low	0
1	Drive High	1
2	Expect Midband	M
3	Expect Valid	V
4	Expect Low	L
5	Expect High	H
6	Hold Window	W
7	Don't Care, HiZ	X

Source Selections for IO Channels

The three source bits that decode to one of the eight states can be selected (using the source select RAM) to come from either the SVM or ADS, or they can be forced to be 0. This selection is up to the programmer. Each of the three bits is individually selected on-the-fly by accessing a source select RAM.

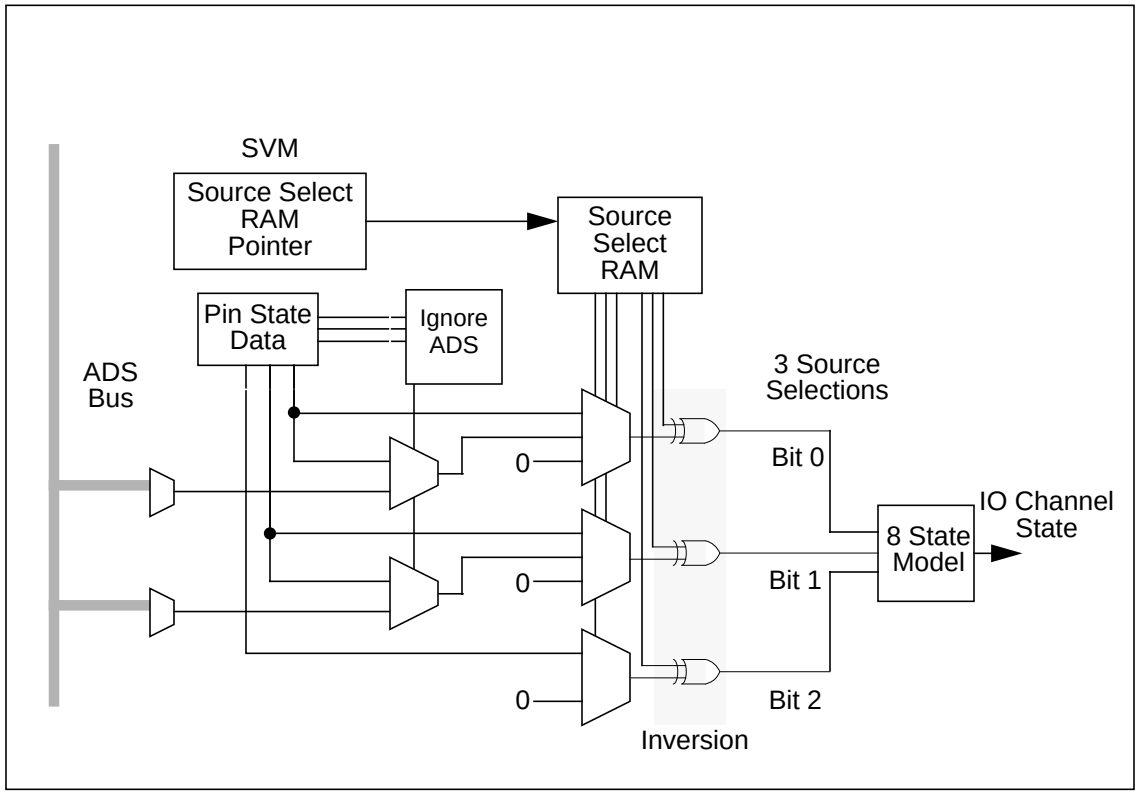


Figure 2.1-31 Source Select for IO Channels

Figure 2.1-31 shows the source select hardware block for IO channels. The hardware shown in the figure (with the exception of the ADS bus) is replicated for every IO channel. Each IO channel can be programmed independently. The values of bits 0, 1, and 2 determine which of the eight states will be selected. (Bit 0 is the LSB.) These bits enter the eight-state model lookup, which maps the three bits to the eight states. The default mapping is shown in Table 2.1-4 on page 2.1.32.

Not all the bits need to come from the same place. Bits 0 and 1 can independently select data from the ADS bus, the SVM (pin state data), or logic 0. (Any bit can be inverted before entering the eight-state model lookup.) If bits 0 and 1 select their data from the ADS bus, they do not have to be the same ADS bus bit. The multiplexers selecting data from the ADS bus are completely independent, permitting one data generator input on bit 0 and the other data generator bit on bit 1 if desired. Bit 2, the MSB, can get its data from either the SVM or logic 0.

The input used for each of the 3 source select bits is determined by the source select RAM. The source select RAM contains 16 locations. Each location specifies which input (ADS, SVM, 0, or their inverts) will be used for all 3 bits. Location 0 is predefined by the job executive. It selects SVM for bit 0, SVM for bit 1, and SVM for bit 2. Locations 1 through 15 are available for the programmer to use as desired. The particular contents of these locations within the source select RAM is

established at load time. The source select pointer, however, can be changed during a pattern generator run, changing the source of any of the three source select bits on-the-fly.

The output of the eight state model lookup hardware is the three bits that define one of the eight possible states the IO channel can be placed in. This data goes to the IO channel formatters. The formatters interpret the bits and places themselves in the specified state. In the default mode, the eight-state model does not change the bits; each bit is passed through unchanged. However, if desired, the eight-state model lookup can be reprogrammed so that a different IO channel state is generated from the three bits that come into the eight-state model lookup table.

If the source select RAM specifies all bits to come from the SVM, and the eight-state model is at its default setting, then the symbol used in the pattern (0,1, m, x, etc.) will set the state of the formatter.

What is often useful is to specify ADS as the source for bit 0 and svm as the source for bit 1 and bit 0. (The source select syntax for this arrangement is: (vm vm ads).) This produces the mapping shown in Table 2.1-5. As is shown in the table, bit 0 will choose between drive 0 and drive 1. With bit 0 coming from the ADS bus, that means the data generator is supplying the data. This is desired. However, note that the data generator also chooses between expect midband and expect valid in the table. This is not desirable. The ignore-ADS bits are used for this. Each one of the eight states can be programmed to ignore the ADS inputs. Also, the ignore-ADS bits are selected by the pattern independent of the source selections. In this way, states that are independent of the ADS data can be reliably selected.

Table1.8.2 on page 1.8.7 provides a list of recommended source selections.

Table 2.1-5 Example IO Channel Source Select Bits Assignment

Bit Assignment			IO Channel State
SVM	SVM	ADS	
Bit 2	Bit 1	Bit 0	
0	0	0	Drive 0
0	0	1	Drive 1
0	1	0	Expect Midband
0	1	1	Expect Valid
1	0	0	Expect Low
1	0	1	Expect High
1	1	0	Hold Window
1	1	1	Don't Care, HiZ

Source Selections for Drive Only Channels

The drive only channels do not use an eight-state model. They do however use source select RAM to determine if the source is the ADS bus, SVM, or logic 0 (or their inverts).

The drive-only channel formatters are divided into two halves, the X half and the Y half. Figure 2.1-32 shows the source select hardware for a drive-only channel. Both halves of the formatters are used when format multiplexing. For non format multiplexing, only the X half is used. Although there are three source select multiplexers, only the first two are used for each half, resulting in a total of four bits used altogether. The X half uses bits 0 and 1, and the Y half uses bits 2 and 3. These bits describe the drive data at d1 and d2 (for X data) and d4 and d5 (for Y data).

The source selection scheme for drive only channels is as follows:

- ❒ In a simple non-muxed drive-only channel, only one source of data is used. Valid data appears from d1 to d2
- ❒ In a format-muxed drive-only channel, two sources of data are used. One bit is used for the first half of the cycle, and the other bit for the second half. Valid data appears from d1 to d2 and from d4 to d5
- ❒ In a drive-only channel that is using both Mach2 and format-muxing, four sources of data are used. Two bits are used in the first half and two bits are used for the second half. Valid data appears at d1, d2, d4, and d5

Depending on the mode of the channel, various bits of the source select are used. Basically, each bit of the source select is associated with an edge:

- ❒ Bit0 is associated with the d1 edge. This bit is used in all modes.
- ❒ Bit1 is associated with the d2 edge. This bit is used only when the channel is in Mach2 mode.
- ❒ Bit2 is associated with the d4 edge. This bit is used only when the channel is in Format-mux mode.
- ❒ Bit3 is associated with the d5 edge. This bit is used only when the channel is in both Mach2 and Format-mux modes.

The source select syntax for the drive only channels is similar to that of the IO channels. An example would be (0 vm ads). The 0 is required for the MSB. This sets the unused multiplexers for both the X and Y half to logic 0. Bits 1 and 3 are set to SVM, and bits 0 and 2 to the ADS bus. Note that the source is set for both the X and Y half. The system can be programmed with separate source selects for the X and Y halves.

The typical syntax for a drive-only channel to be used as a ***clock*** is (0 0 vm). (For mach2 operation, the clock should be (0 vm vm).) Only the X half of the channel is used and the drive data comes from the SVM. A symbol of 0 or 1 in the pattern would program the clock to be low or high.

The typical syntax for a drive-only channel to be used for ***address*** is (0 0 ads). This causes the ADS input to drive the channel. The address driver will then follow the address generated by the address counters inside the memory pattern generator.

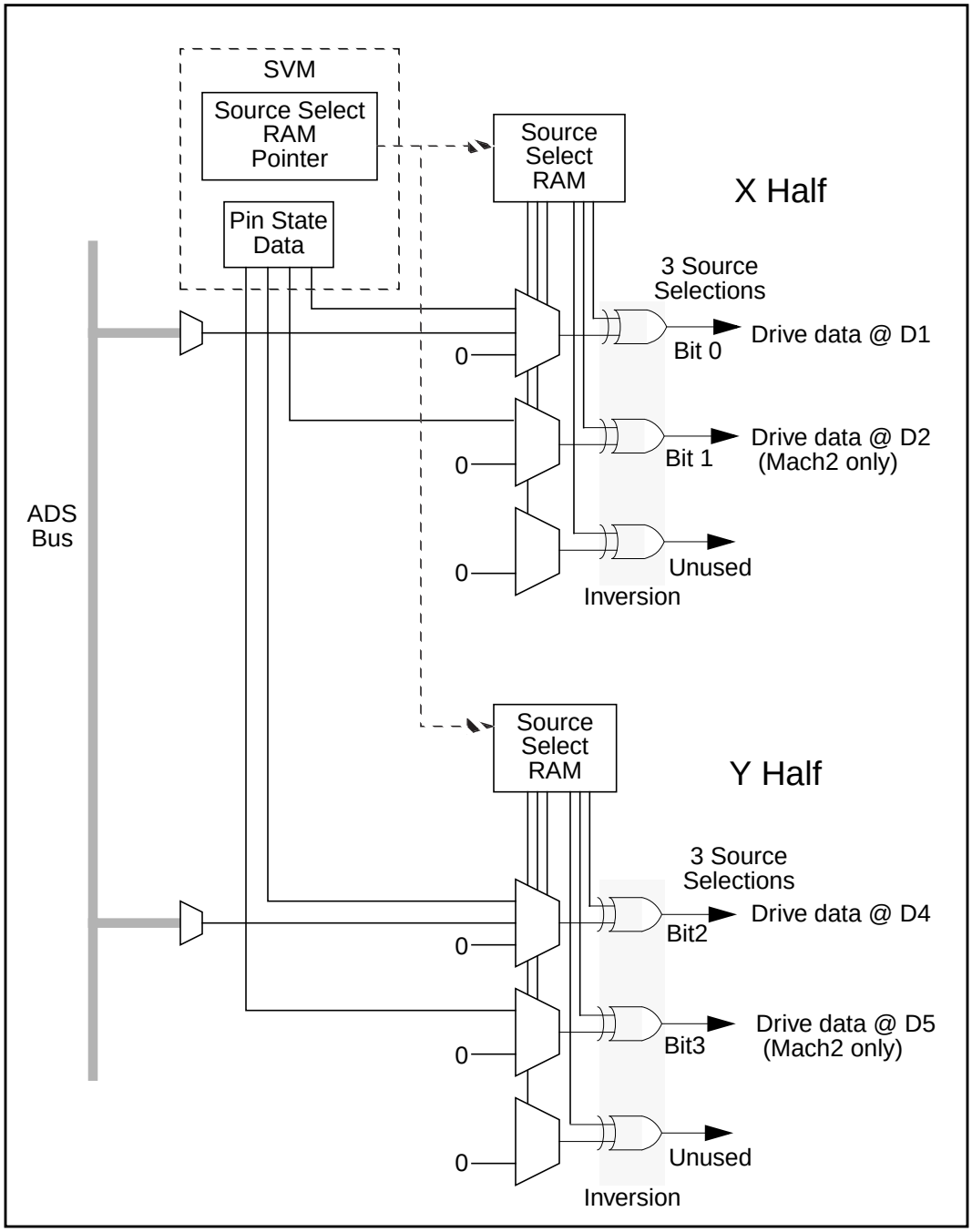


Figure 2.1-32 Source Select for Drive-Only Channels

Source Select Programming for SCIO Applications

Those applications that require both drive and compare in the same cycle will use same cycle IO. The eight-state model described on page 2.1.32 does not provide SCIO operations. To achieve SCIO, the IO channel formatters are put into SCIO mode. This mode changes the eight-state model inside the formatters enabling SCIO.

The commands for enabling the modes are:

`:scio valid` or
`:scio midband`

These commands appear in the waveadr database. Once one of these modes is selected, a new eight-state is selected as shown in Table 2.1-6 below.

Table 2.1-6 Eight-State Model Valid [a]

Bit Assignments			State	IO Channel State	Programming Symbol
Bit 2	Bit 1	Bit 0			
0	0	0	0	drive 0, expect 0	l
0	0	1	1	drive 0, expect 1	h
0	1	0	2	drive 0, expect valid (midband)	v
0	1	1	3	drive 0, expect don't care	x
1	0	0	4	drive 1, expect 0	L
1	0	1	5	drive 1, expect 1	H
1	1	0	6	drive 1, expect valid (midband)	V
1	1	1	7	drive 1, expect don't care	X

a. If :scio midband is selected, then midband replaces valid as shown in parenthesis in this table.

Once the desired SCIO mode is selected, the source for the bits must be selected. Examining Table 2.1-6 suggests what the proper arrangement should be. Bit 0 selects between expect 0 and expect 1. This choice is a data generation choice. Therefore bit 0 should have the ADS bus as its source. Bit 1 selects between data and don't care or valid. This distinction is programmed in the pattern. Therefore bit 1 should be sourced from the SVM. Bit 2 chooses between drive 0 and drive 1. This is a data generator consideration. Therefore bit 2 should be sourced from the ADS bus. Therefore the appropriate source selects for SCIO would be (ads, vm, ads). However, Figure 2.1-31 on page 2.1.33 shows that bit2 cannot be sourced from the ADS bus.

The solution is quite simple. Use the eight-state model lookup table to remap the bit assignments to swap bit 2 and bit 1. This is shown in Table 2.1-7.

Table 2.1-7 SCIO Eight-State Remapping

Bit Assignments (Standard)			State	Bit Assignments (remap)			State
Bit 2	Bit 1	Bit 0		Bit 2	Bit 1	Bit 0	
0	0	0	0	0	0	0	0
0	0	1	1	0	0	1	1
0	1	0	2	1	0	0	4
0	1	1	3	1	0	1	5
1	0	0	4	0	1	0	2
1	0	1	5	0	1	1	3
1	1	0	6	1	1	0	6
1	1	1	7	1	1	1	7

In Table 2.1-7, bits 2 and 1 in the standard mapping were interchanged in the remap. With this remap, the source selects can be set to (vm, ads, ads). The eight-state lookup table would be remapped as:

0 to 0

1 to 1

2 to 4

3 to 5

4 to 2

5 to 3

6 to 6

7 to 7

The remapping is programmed in the srcadr database. For example,

```
(srcset   abc
   (data-map-form
          (IOS)
          (#o76325410)
   )
   (ignore-ads-form
          (IOS)
          (#o0)
   )
)
```

Since either drive or detect data is needed for all eight states, the ignore ADS bits cannot be used. Therefore, to prevent the data generator output from switching between valid and don't care states, two source selects should be used.

For states 0, 1, 4, and 5 use a source select of:

(vm ads ads)

For states 2, 3, 6, and 7 use a source select of:

(vm ads vm)

IO Channel Fanout to the Test Head

In the J997 and J993, each test head is driven by up to 9 WaveGen Boards (also known as a Hexadecimal Channel Slice or HCS board). Each slice of these WaveGen boards generate signals for one IO channel. There are also eight IO channels per IO channel board. This means that each IO channel WaveGen board supports one IO channel board.

Eight IO slices per WaveGen board (and eight IO channels per IO channel board) times nine boards each gives a total of 72 IO channels per test head.

WaveGen (DCS) boards are located in the mainframe, and IO channels are located in the test head. Each IO slice has independent timing, format, and error logic control.

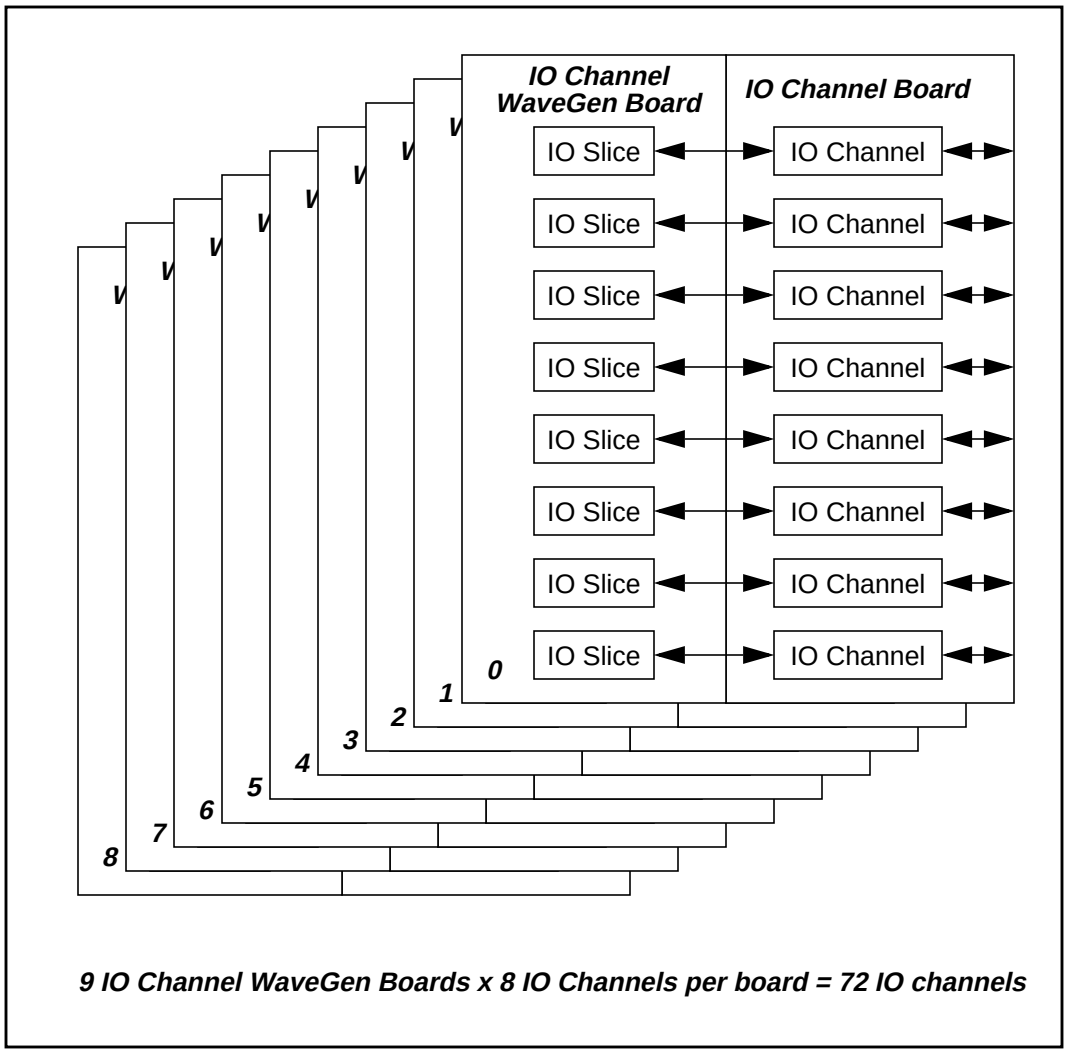


Figure 2.1-33: J997 and J993 IO Channel Fanout

In the J994, each test head is driven by up to 9 WaveGen Boards (which are knows as Hex Channel Slice or HCS boards on the J994). Each J994 WaveGen board contains 16 IO channel slice. Each slice off these WaveGen boards generate signals for a single IO channel. Like the other members of the J990 series, there is

a one-to-one relationship between the slices on the WaveGen board and the slices on the IO Channels.

WaveGen (HCS) boards are located in the mainframe, and IO channels are located in the test head. Each IO slice has independent timing, format, and error logic control.

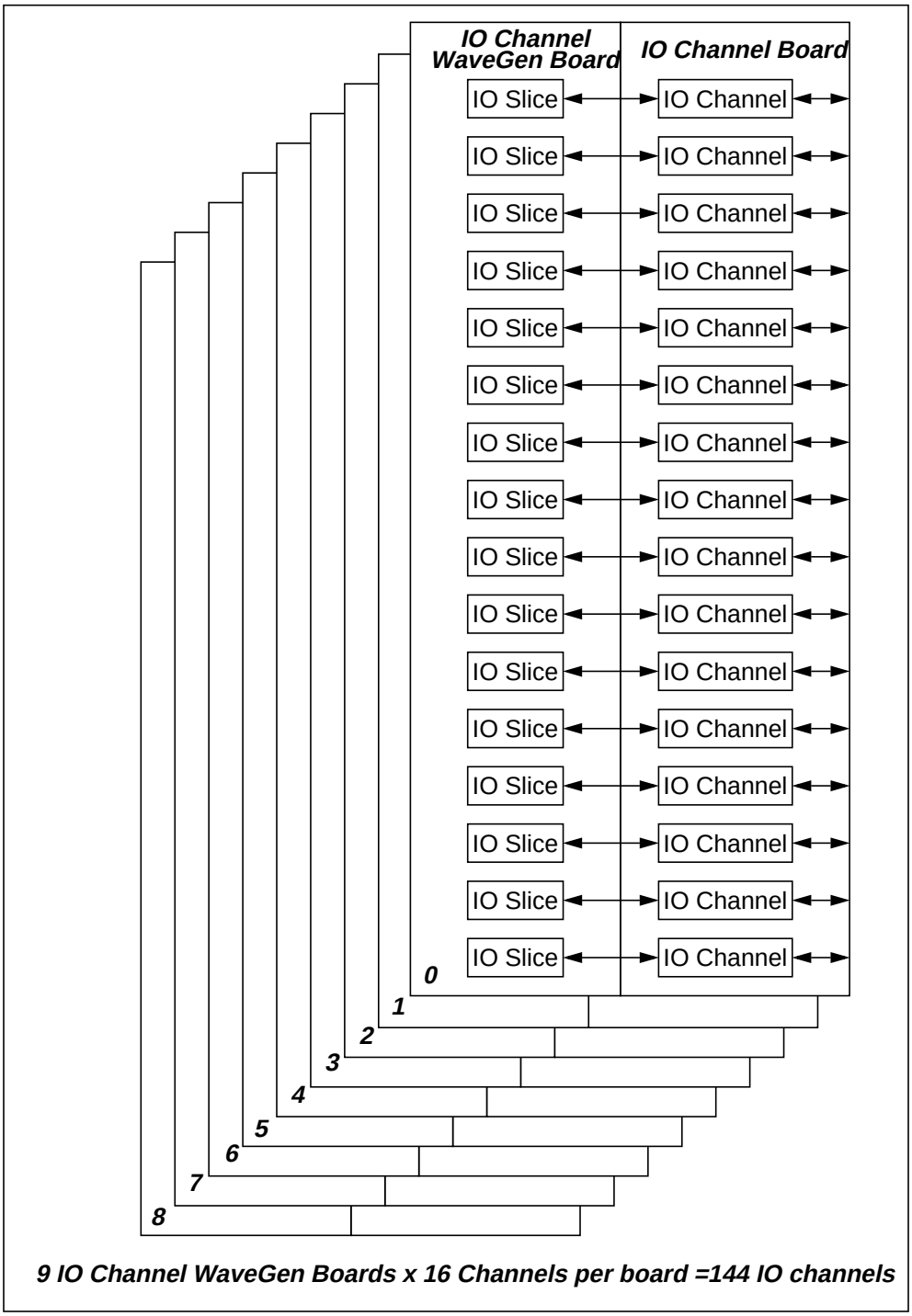


Figure 2.1-34: J994 IO Channel Fanout

Drive-Only Channel WaveGen Fanout (J997, J994, and J993)

For all members of the J990 series, there are up to 40 programmable drive-only *functions*. A function is an individually programmable timing and format signal, usually applied to individual input pins on a device. Drive only functions are typically used for address and clocks.

Eight functions are located on each of five Drive-Only Wavegen boards (also known as Memory Channel Slice or MCS boards).

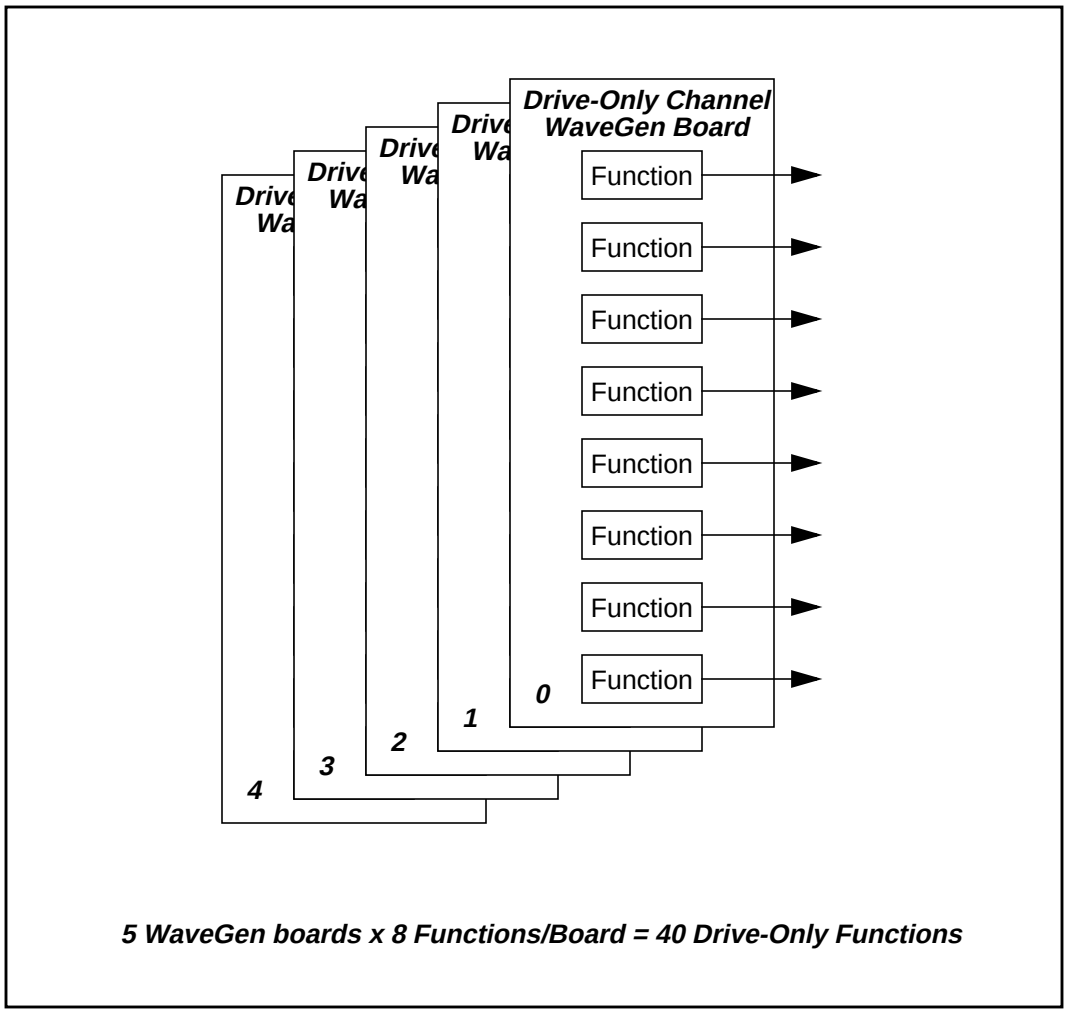


Figure 2.1-35: Drive-Only Channel Functions

Each WaveGen board supports four channel cards.

Drive-Only Channel Fanout to the Test Head (J997 and J993)

The fanout between the drive-only WaveGen boards, located in the mainframe, and drive-only channel cards in the system, vary among the various members of the J990 series. Figure 2.1-36 shows the fanout for J997 and J993 systems.

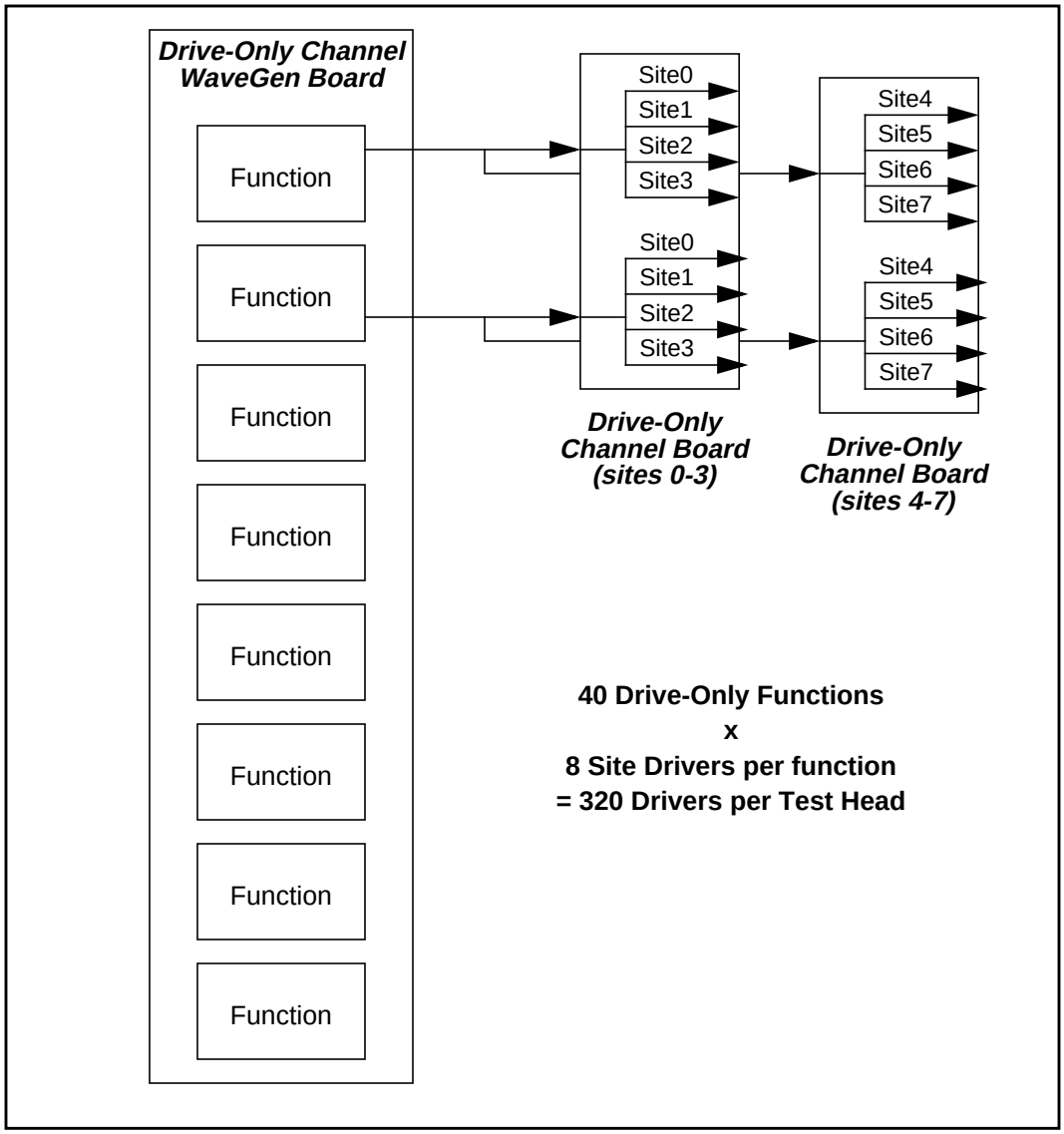


Figure 2.1-36 Drive-only Channel Cards Support 2 Functions For 4 Sites (J997, J993)

It is important to note that each function is split in half. Four site drives are located on the drive-only channel board for sites 0 through 3, and four site drivers with the same function are located on the drive-only channel board for sites 4 through 7. A second function from the WaveGen board will supply the remaining site drivers on the selected channel boards. This arrangements permits some applications not requiring many sites or not needing many drive functions to be implemented using fewer channel boards, resulting in a less costly test head stuffing.

Drive-Only Channel Fanout to the Test Head (J994)

Figure 2.1-37 shows the fanout for J994 systems. Channel cards for the J994 are different from those used in the J997 or J993. J994 channel cards contain 16 drivers per card; twice as many as the other systems.

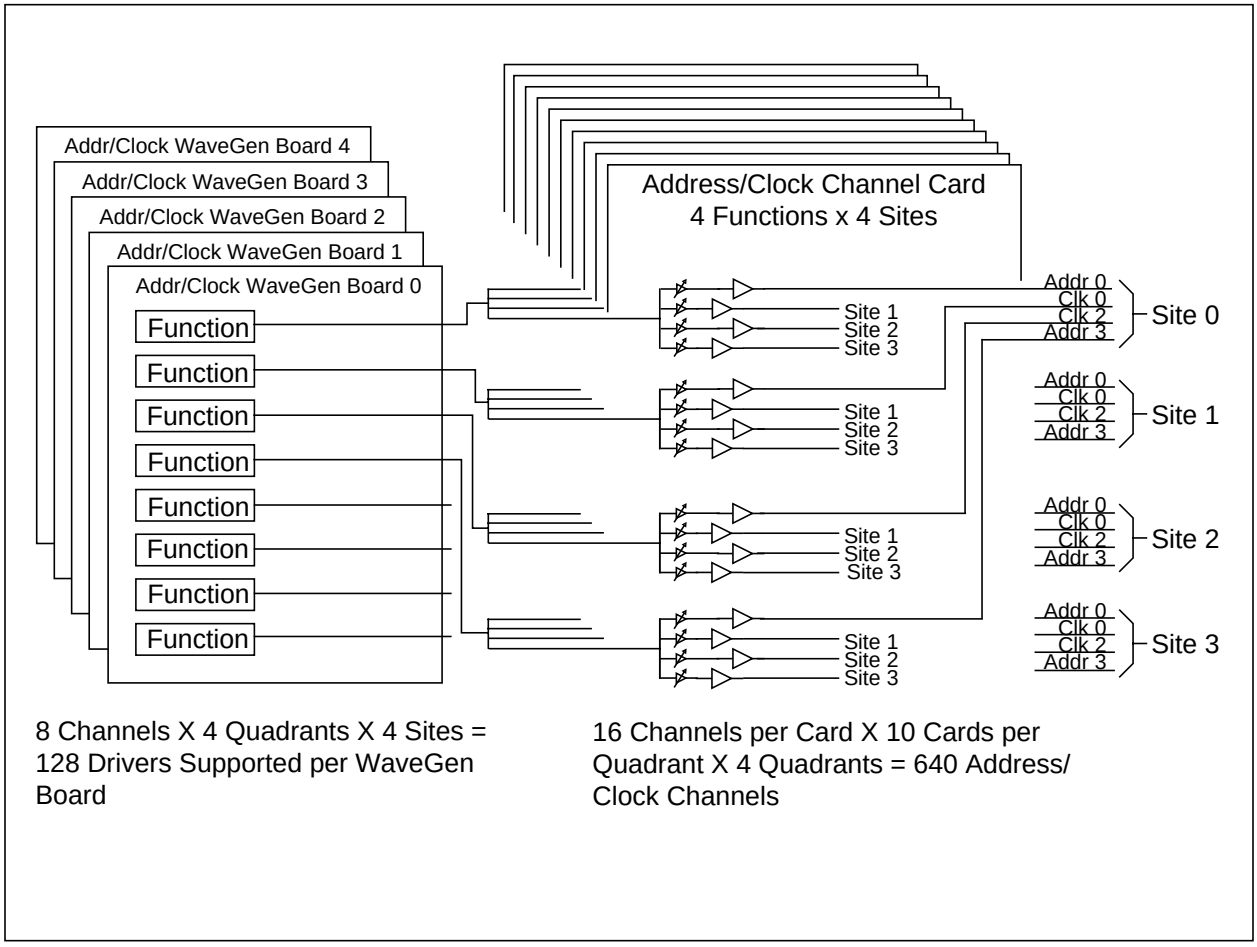


Figure 2.1-37 Drive-only Channel Cards Supports 4 Functions For 4 Sites (J994)

Just as with the other members of the J990 series, the J994 splits the function among multiple channel cards (four cards for each function). This arrangement may allow fewer channel cards to be used with some applications.

Drive-Only Channel Configuration for Up to 8 Sites (J997 or J993)

Each drive-only channel board supports two functions for four sites, or eight drive signals. The signal fanout actually supports eight sites, as each function is routed to two channel boards.

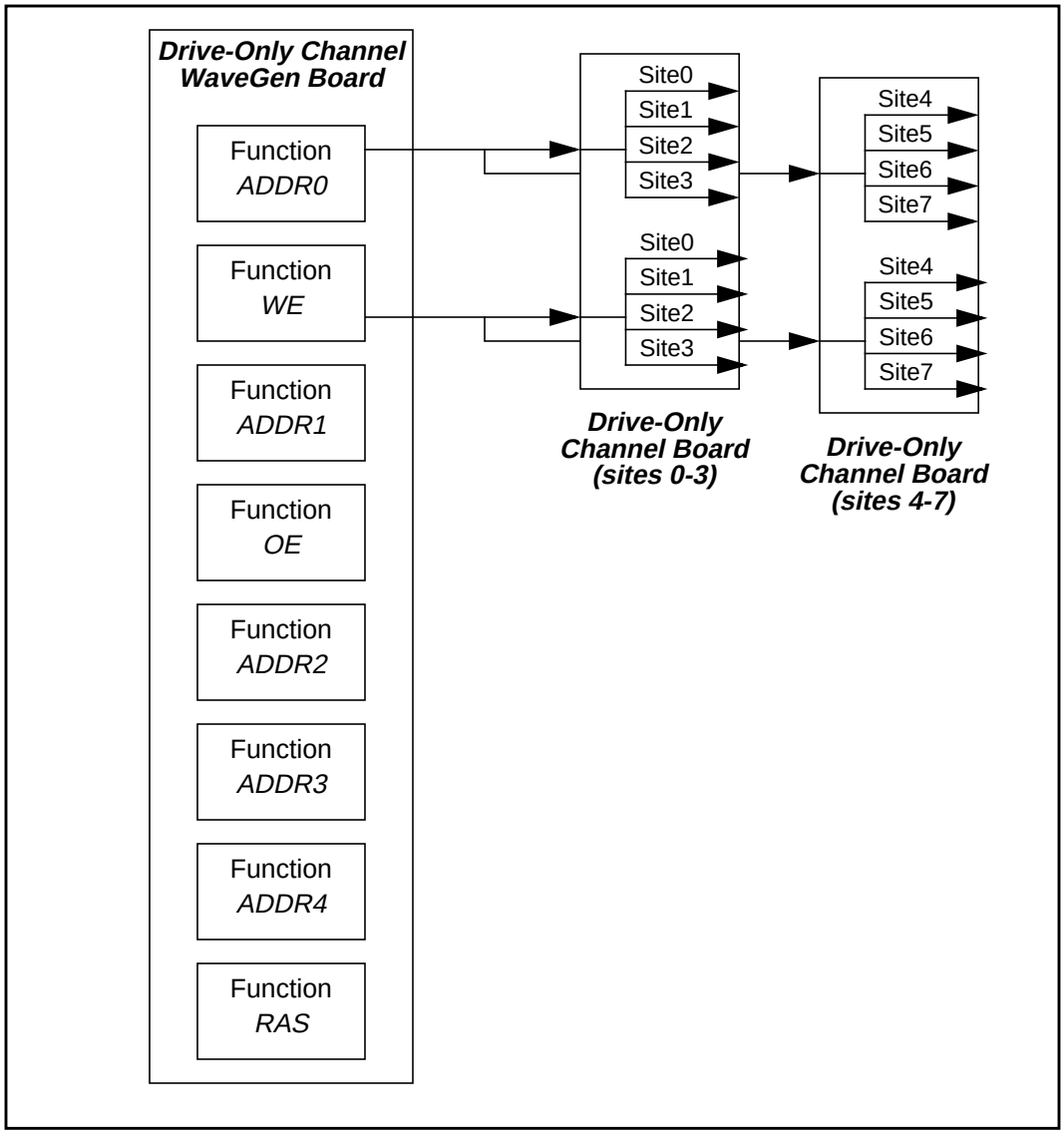


Figure 2.1-38: Example Drive-Only Function Assignments for Eight Sites

When eight sites or less are being tested on the J997 or J993, there are 40 drive-only functions available.

Drive-Only Channel Configuration for More than 8 Sites (J997 or J993)

When more than eight sites are being tested, 20 functions are available. In this case, half of the 40 functions are providing the same signal as the other half.

The assignment of which channels will have the same function is up to the programmer. There is no hardware requirement in this area. This choice can be made with the interface requirements in mind.

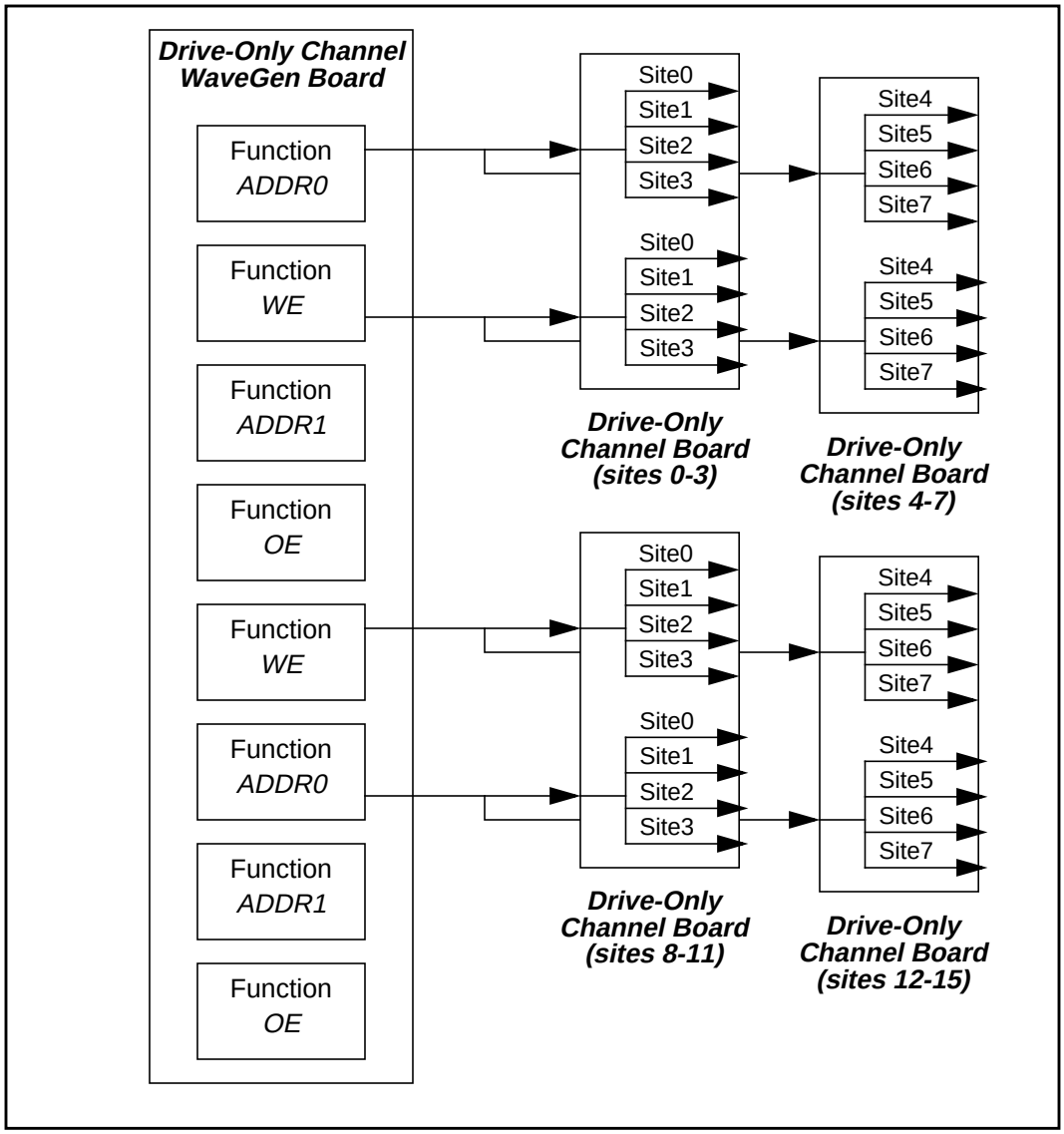


Figure 2.1-39: Example Drive-Only Function Assignments for Sixteen Sites

The selections of which site drivers go to each device site is up to the programmer. For example, the site 0 driver signals can be connected to device site 5. This means the definition of the site is controlled by the software, not the hardware.

Drive-Only Channel Configuration for Up to 16 Sites (J994)

Since each WaveGen Function is fanned out to 16 channel card drivers, each function is independent for a test head with 16 or fewer sites. Up to 40 functions can be programmed.

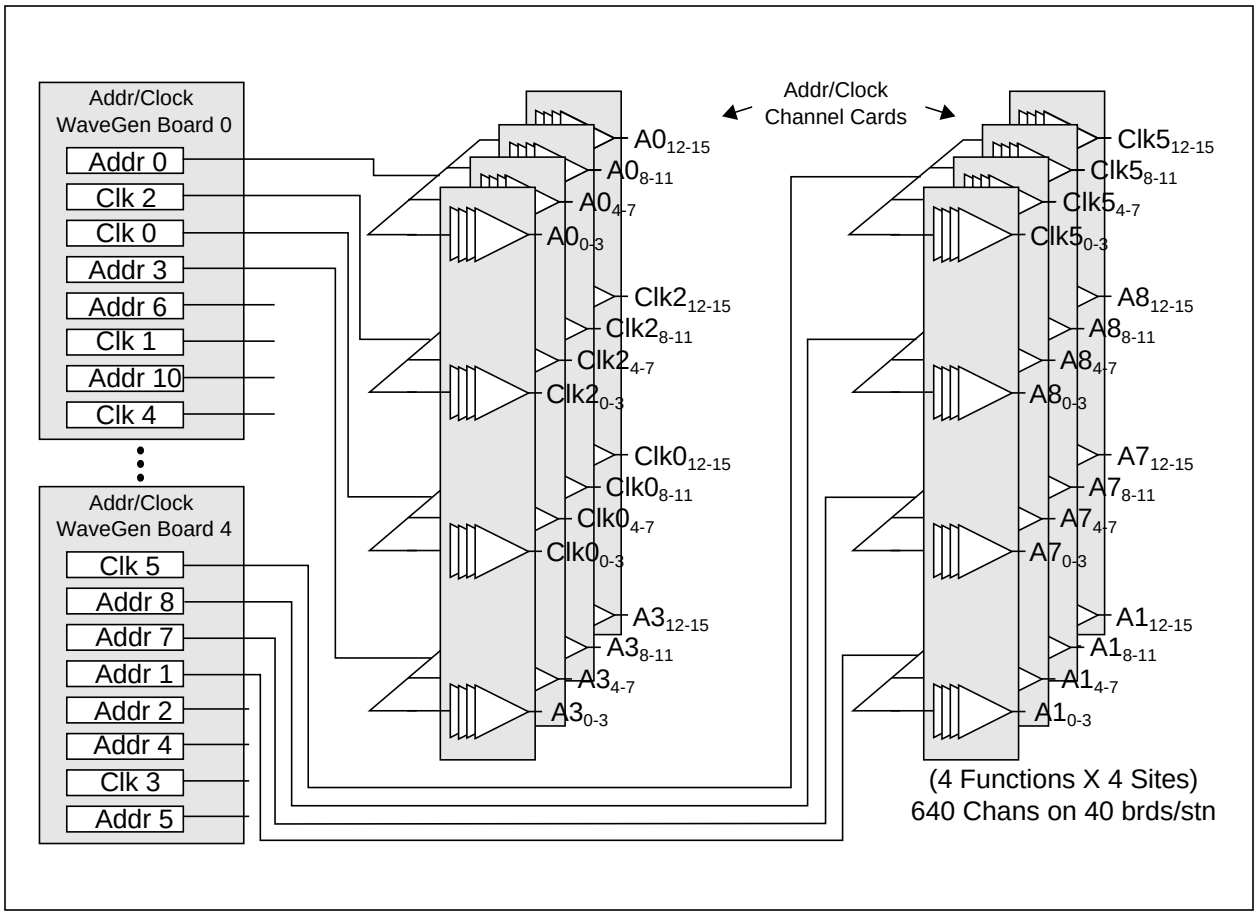


Figure 2.1-40: Example Drive-Only Function Assignments for Sixteen Sites

Drive-Only Channel Configuration for More than 16 Sites (J994)

If more than 16 sites are needed per test head, the number of available WaveGen functions are reduced to 20. Two WaveGen functions will be programmed identically to cover the additional sites.

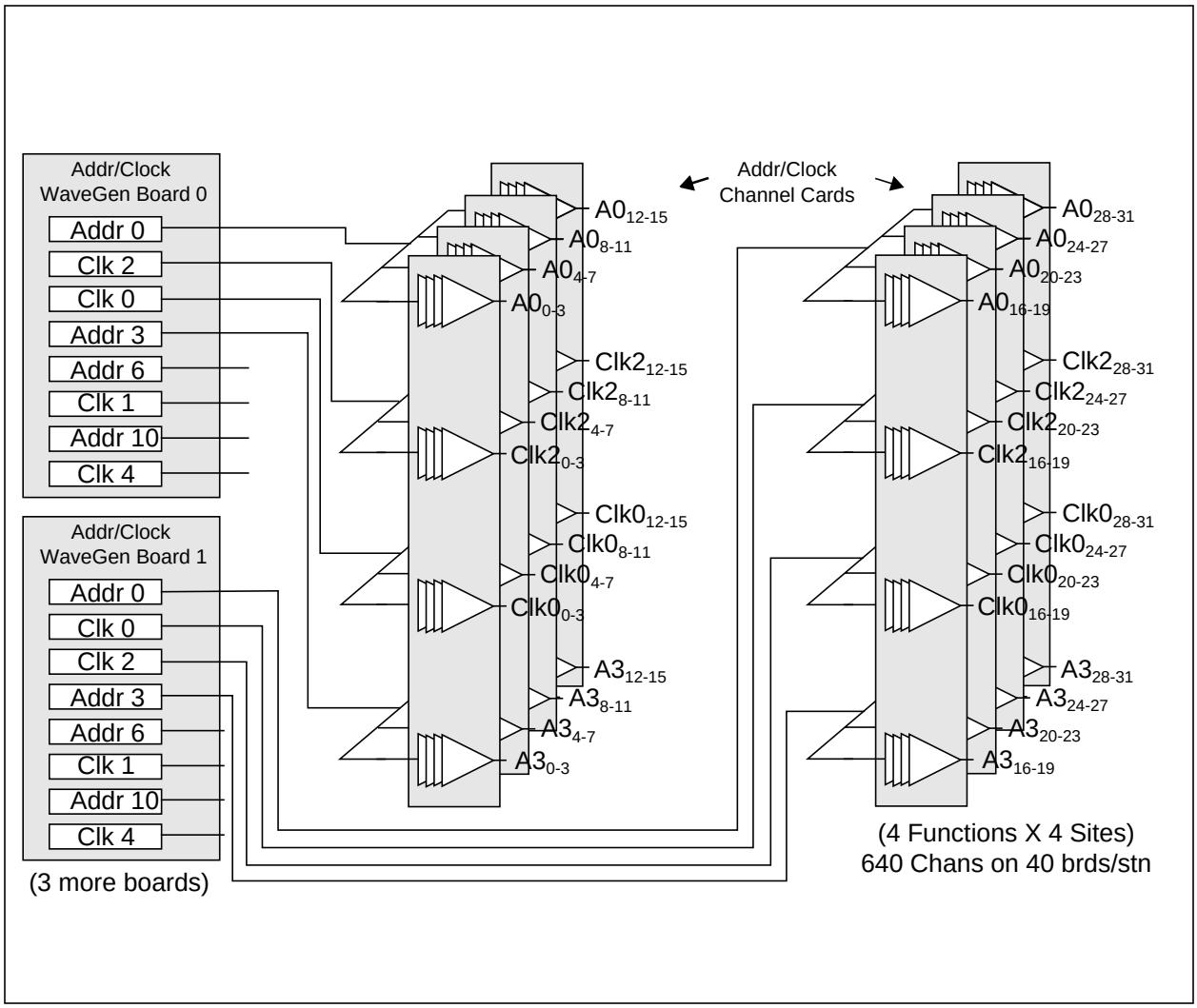


Figure 2.1-41: Example Drive-Only Function Assignments for 32 Sites

Drive-Only Channel Functional Description

Each Drive-only channel can be programmed independently. This includes:

- ❒ Driver levels: Drive Voltage High (Vih), Drive Voltage Low (Vil)
- ❒ Device-Under-Test connect to isolate device pin
- ❒ Functional connection to driver-comparator circuit
- ❒ Measurement connection to Pin/Site PMU, System PMU, and Cal matrix

The J997 drive-only channel has the Pin PMU, just like the IO channel. The J994 system has a site PMU which can connect to each drive-only pin that is part of a particular hardware site. The system PMU is available on all systems; however, for the J994 only, the system PMU can only be connected to the drive-only channels through the FET matrix — the relay matrix cannot be used.

The channel is designed for a 50 ohm environment to assure signal fidelity. The channel drive data changes on-the-fly while the patgen is running, and all other functions are set using the databus.

The driver range is selected using the system databus, and should be programmed prior to starting the patgen. One of the following ranges must be selected:

- ❒ -4V to +5V
- ❒ -3V to + 6V
- ❒ -2V to +7V

The system should also be calibrated in the same range the channel will be used in. There is only one range per test head. This range applies to drive-only and IO channels.

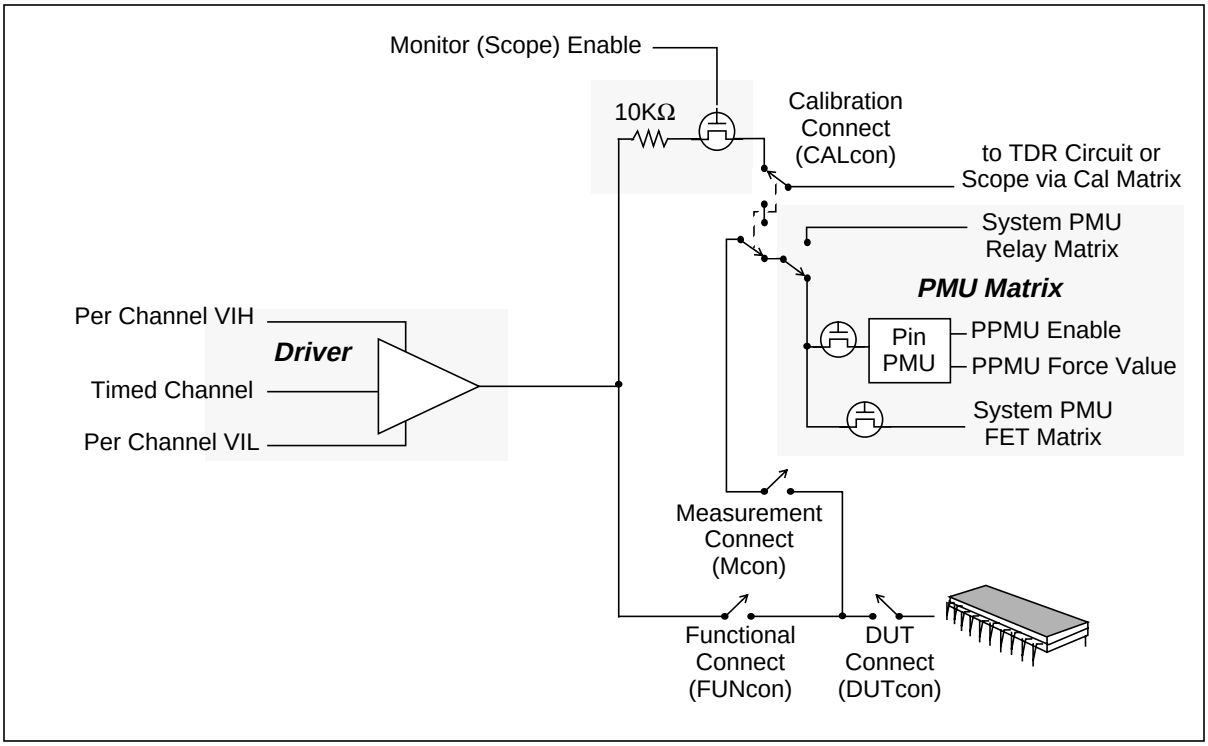


Figure 2.1-42: Drive-Only Channel Block Diagram

IO Channel Functional Description

Each IO channel can be programmed independently. This includes:

- ❒ Driver levels:

  Drive Voltage High (Vih),
  Drive Voltage Low (Vil)

- ❒ Comparator levels:

  Compare Voltage Low (Vol), Compare Voltage High (Voh),
  Current Load High (Ioh), Current Load Low (Iol),
  Termination Voltage (Vt),
  Voltage Clamp Low (Vcl), Voltage Clamp High (Vch)

- ❒ Device-Under-Test connect to isolate device pin
- ❒ Functional connection to driver-comparator circuit
- ❒ Measurement connection to Pin PMU, System PMU, and Calibration matrix

The channel is designed for a 50 ohm environment to assure signal fidelity. The channel drive data and HiZ/Load enable changes on-the-fly while the patgen is running, and all other functions are set using the databus.

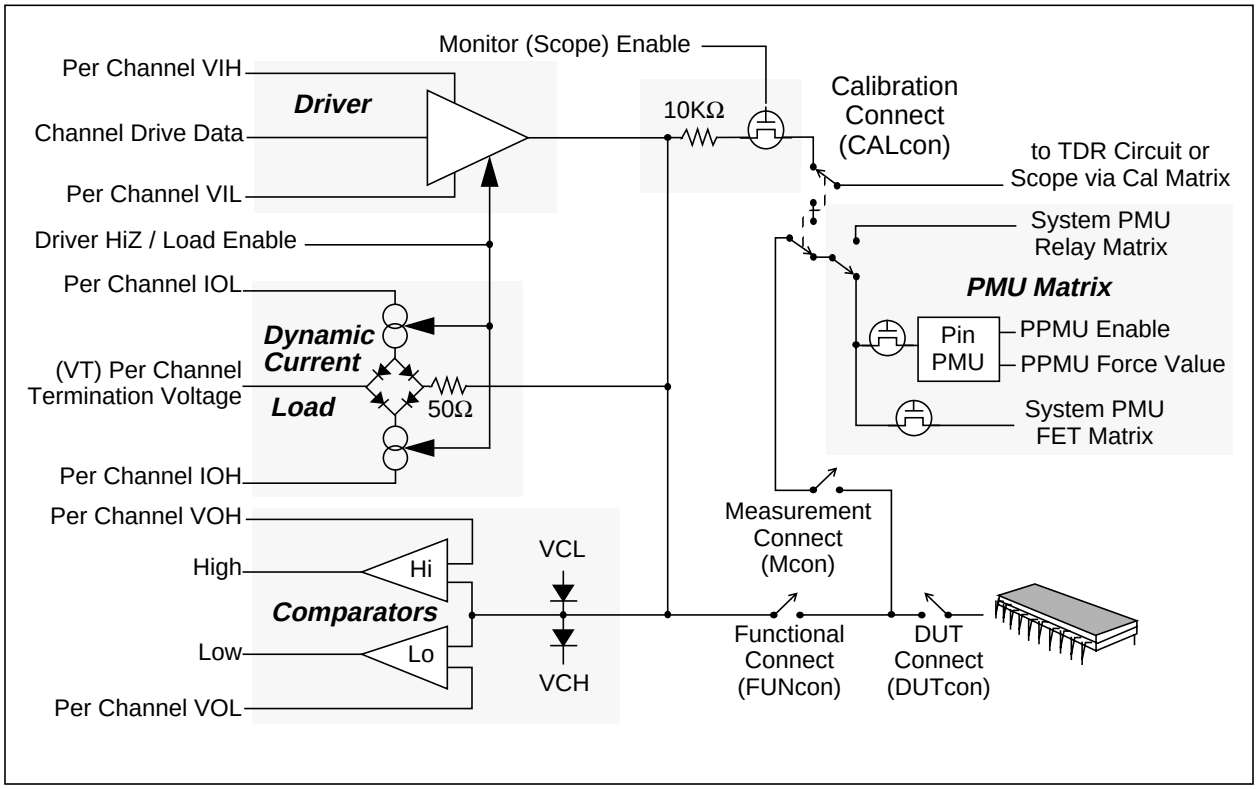


Figure 2.1-43 IO Channel Block Diagram

Dynamic Current Load

Each IO channel has an individually programmable current load. The IOH and IOL levels can be programmed between 300 µA and 50 mA. This circuitry allows the device loading to be easily changed from test to test.

Impedance Lock

Modern semiconductors have rise and fall times fast enough to warrant a transmission line environment, but are usually designed with low impedance outputs and high impedance inputs. The result is ringing, sometimes severe enough to affect device operation. To quiet the input to the comparators, a pair of clamp diodes have programmable voltage references (Vcl, Vch). The programmer has the choice of programming these so they have no affect, or setting Vcl and Vch close to the actual device voltage levels and using the clamping circuit.

Pin Index Keys (PIKs) and the PIK Map RAM

To control individual and parallel channel programming, the system has a channel map RAM, which selects a channel or set of channels for each programmed RAM location. The address into this RAM is called a Pin Index Key (PIK). In essence, a PIK is an address into the channel map RAM (also known as the PIK map RAM).

The lower addresses of the RAM are mapped to correspond directly to the channel numbers (PIK 0 accesses channel number 0, PIK 1 accesses channel number 1, etc.). Upper addresses (PIKs) are programmed (by the job executive) to access multiple channels in parallel, based on the pin groups in the test program. This permits the parallel programming of channels that have the same information.

For example, address pins on a device are commonly programmed to have the same timing. Assuming the test program programs the address pins as a group, the timing for the address pins will be loaded in parallel. These definitions are created in the pinadr database.

The device voltage sources are not addressed using PIKs. Although the pinadr database will permit a group pin symbol to include both VS and PIK symbols, this would force the pins to be addressed individually, slowing execution.

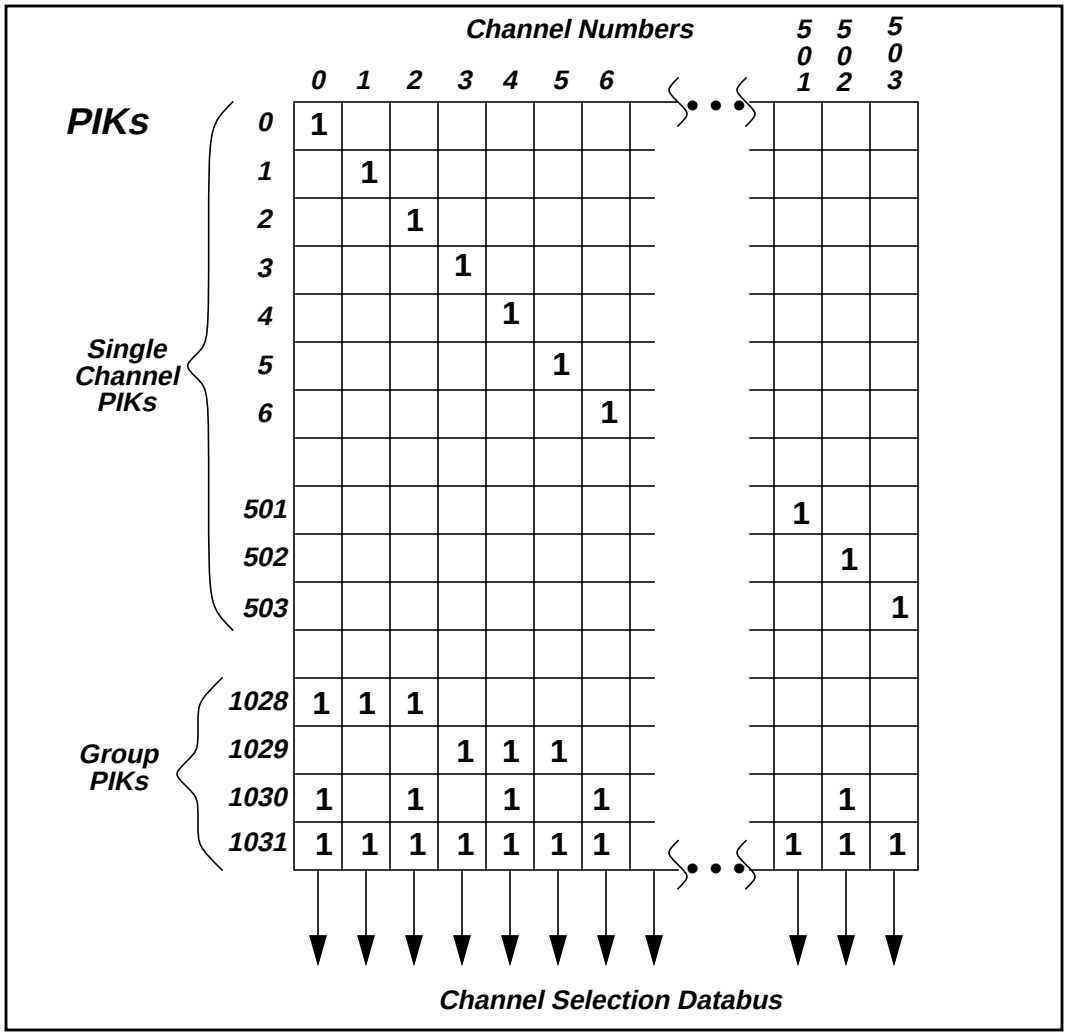


Figure 2.1-44: PIK Mapping Illustration (the group PIKs shown are not accurate numbers)

Voltage Sources

Power for the DUTs is provided by the voltage sources. The voltage sources include programmable current limits and current monitors to measure output current. The voltage sources are supplied in groups of eight, and up to sixteen are available per test head. Each voltage source can supply ± 1A at up to ± 10V and 0.25 A at up to 20V. Up to 8 voltage sources can be wired in parallel for greater currents.

Six current ranges are available for making measurements:

Range	***Resolution***
+/- 1A	50 µA
100 mA	5 µA
5 mA	250 nA
1 mA	50 nA
100 µA	5 nA
10 µA	500 pA

Each voltage source has two voltage values, main and alternate. The pattern generator can control which is used for VBUMP testing. If bump testing is not being used, the main and alternate values can be set equal.

Each VS board also contains a bias supply. The bias supply can be used at the test head for any low current (20 mA) biasing.

Pin Parametric Measurement Unit (Pin PMU)

Each IO and drive-only channel on the J997 has a Pin PMU, typically used for leakage measurements and contact/continuity tests. Each Pin PMU has independent force values, and can make measurements in parallel. The measured result can then be read back from each independent channel. In addition, each Pin PMU can be programmed to have the test limits (in hardware). When a measurement is taken, the hardware will provide a measured value and a pass/fail result for each channel. A ganged pass/fail result can also be read back when multiple pins are accessed simultaneously.

The Pin PMU controller is a central resource shared by all of the Pin PMUs. It controls the range selection and the force current/measure voltage vs. force voltage/measure current selection, which globally controls all Pin PMUs.

There is one voltage range available, which applies when ***forcing or measuring voltage***.

Range	***Resolution***
-4 V to +8 V	0.5 mV

However, it is possible to force voltage up to +12 V, if using the +/- 2 mA current range and the current does not exceed 200 uA.

When ***forcing current***, three current ranges are available:

Range	***Resolution***
+/- 200 uA	40 nA
+/- 2 mA	400 nA
+/- 50 mA	5 uA

The 50 mA is provided by the current loads on the IO channels and is not available on the drive only channels.

When ***measuring current***, five current ranges are available:

Range	***Resolution***
+/- 200 nA	100 pA
+/- 2 uA	10 nA
+/- 20 uA	10 nA
+/- 200 uA	100 nA
+/- 2 mA	1 uA

Since there is a single pin PMU controller, all pins have the same range selected.

For the job executive software, a ganged test will generate a single pass/fail result (0 = pass, 1 = fail) for the pin list being tested. A fail will occur if any pin in the list exceeded the programmed limits. This is a hardware readback.

A sequential test will perform a parallel measurement on the specified pin list, then the measured values are logged individually in sequence.

System Parametric Measurement Unit (System PMU)

There is also a system PMU, which can be used to supply high current (up to 200 mA). It can be connected to any pin in the test head, using either a FET matrix or a relay matrix; however, for the J994 only, the system PMU can only be connected to the drive-only channels through the FET matrix — the relay matrix cannot be used. The FET matrix provides very fast connections, while the relay matrix allows more current to the pin. There is a single system PMU.

The system PMU may be used to provide an different force function than the pin PMU. For example, if the pin PMU is programmed to force voltage, the system PMU can force current.

Depending on whether the FET matrix or relay matrix is used, different ranges are available.

Using the Relay Matrix

Three ranges are available for ***voltage forcing and measuring***:

Range	***Resolution***
+/- 2 V	100 μV
+/- 20 V	1 mV
+/- 40 V	10 mV

Six ranges are available for ***current forcing and measuring***:

Range	***Resolution***
+/- 2 μA	100 pA
+/- 20 μA	1 nA
+/- 200 μA	10 nA
+/- 2 mA	100 nA
+/- 20 mA	1 μA
+/- 200 mA	10 μA

Using the FET Matrix

Two ranges are available for ***voltage forcing and measuring***:

Range	***Resolution***
+/- 2 V	100 μV
- 4 V to + 18 V	1 mV

Five ranges are available for ***current forcing and measuring***:

+/- 2 μA	100 pA
+/- 20 μA	1 nA
+/- 200 μA	10 nA
+/- 2 mA	100 nA
+/- 20 mA	1 μA

When forcing current, a programmable current clamp is available, which will limit the voltage on the pin. This is useful when an open circuit condition occurs, and the voltage is to be limited. It can be programmed from 1 V to 40 V, with 100 mV

resolution.

Patgen Sync Conditions

The central FVM circuit also controls the pattern sync conditions. The sync signal is a calibrated signal, with nominal accuracy relative to the driver signals. The sync pulse is generated when all sync conditions are met for a cycle. These programmable sync conditions include:

- ❒ SVM address
- ❒ SVM accumulator value (used for loop and repeat counters)
- ❒ cycle counter
- ❒ X and Y address seen by the address scrambler (physical/array address)
- ❒ X and Y address seen by the data generator
- ❒ data generator bits (prior to topo inversion)

Some or all of the sync conditions can be used. Each condition is enabled separately. The sync is available at the test head for connecting to an oscilloscope and can be used to control FVM capture.

Chapter 2.2: Timing Guidelines

Introduction

This chapter illustrates various timing formats used by J990 Series Test Systems. Use it as a guideline when developing your test programs. It applies to all systems except as noted within the text.

General Timing Guidelines

The following rules provide guidance in programming timing edges.

1. To guarantee that a pulse will occur on the device pin, timing values must be programmed to allow enough time to meet the minimum pulse width (or duration) for the voltage swing. The following table provides the minimum pulse widths.

Table 2.2-1: Minimum Pulse Width

Amplitude (volts)	Minimum Pulse Width (ns) at 50% points
2.0 V	2.496 ns (≈ 2.5 ns)[a]
3.0 V	2.976 ns (≈ 3.0 ns)[a]
5.0 V	5.984 ns (≈ 6.0 ns)

a. For the J993-01, minimum pulse width is 4.0 ns.

The following example shows representative waveforms.

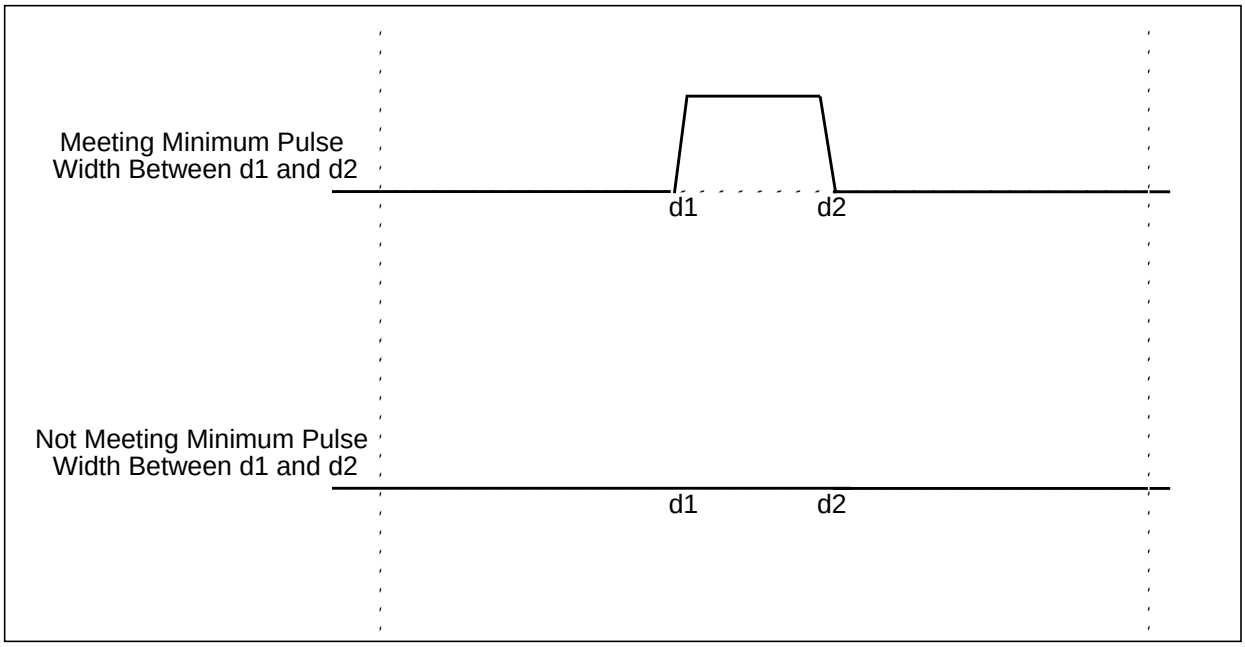


Figure 2.2-1. Minimum Pulse Widths

2. Edges may not cross over like edges.
(That is, d1 may not cross a subsequent d1, etc.)

The following figure provides an example.

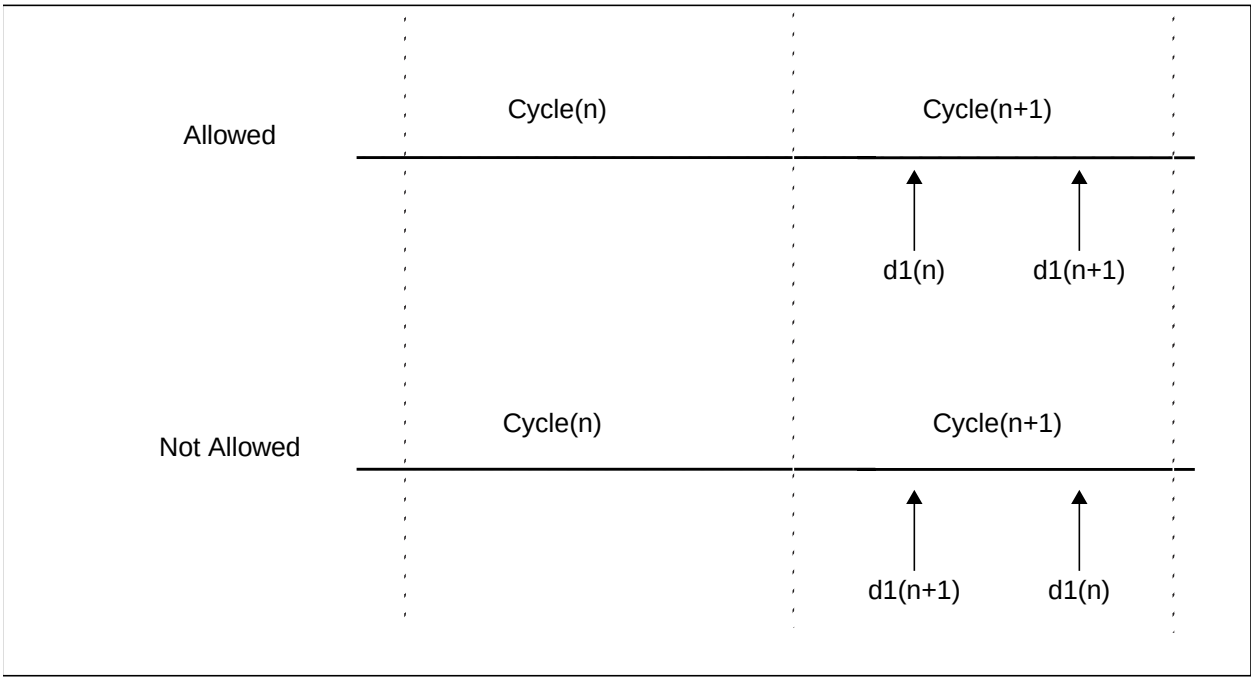


Figure 2.2-2. Edges May Not Cross Over Like Edges

3. While using the 60MHz pattern generator mode, an edge should be at least 20ns apart from a like edge. This limit can be reduced to 17ns if fast-regeneration mode is used.

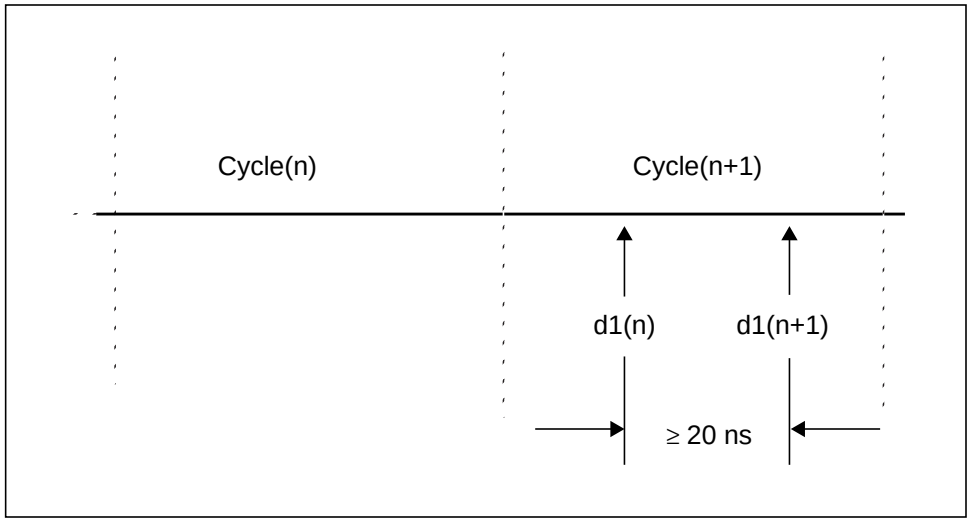


Figure 2.2-3. Edges Should be 20ns Apart (60 MHz Mode)

4. While using the 100MHz pattern generator mode on a J997, an edge should be at least 5ns apart from a like edge in the next cycle. In addition, an edge should be at least 20ns apart from a like edge that is two cycles apart. Again, this limit can be reduced to 17ns if the system is calibrated with fast-regeneration mode.

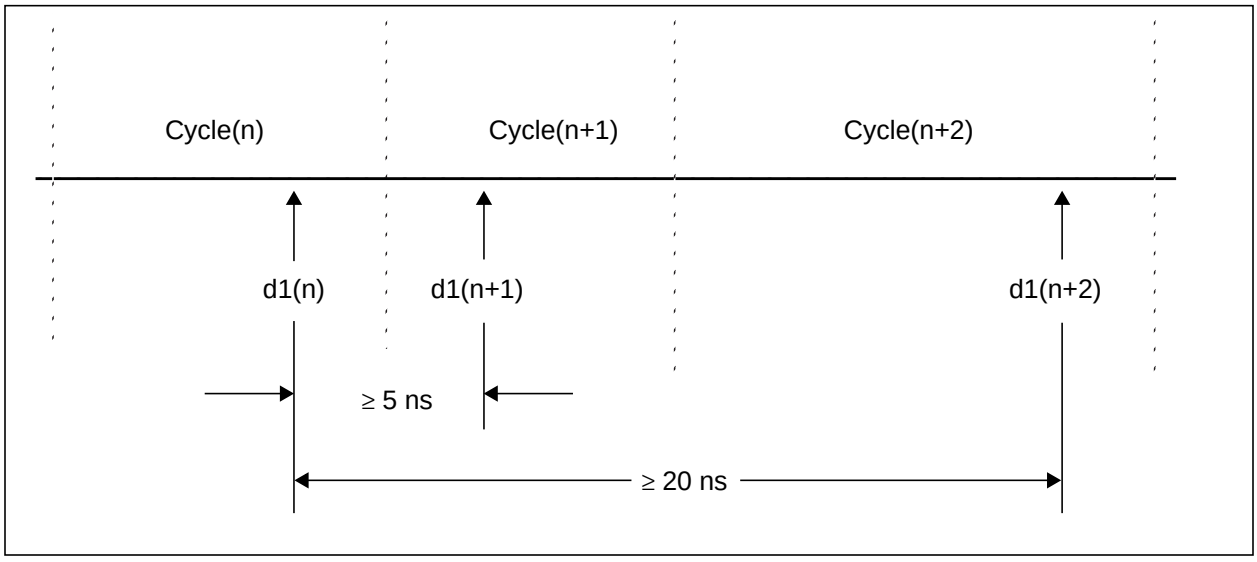


Figure 2.2-4. Edges Should be 5ns Apart (100 MHz Mode)

5. The d2 edge of cycle(n) should not be within 5ns of the d1 edge in cycle(n+1). In addition, for address/clocks in address mux mode, the d5 edge of cycle(n) should not be within 5ns of the d4 edge of cycle(n+1). For 100 MHz, the d5 edge of cycle(n) should not be within 5ns of the d4 edge of cycle(n+2).

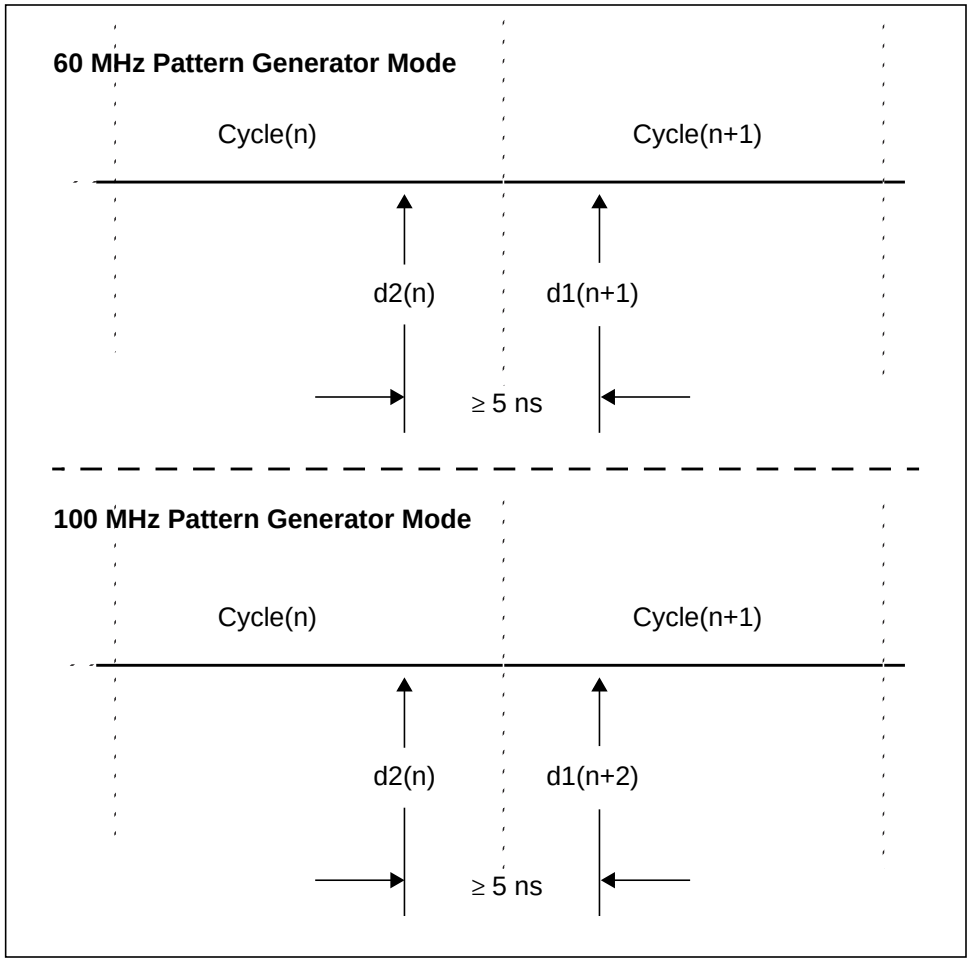


Figure 2.2-5. The Minimum d2 to d1 Time

6. The minimum compare to drive time for the I/O channels is 4ns plus the round-trip time of the DIB. This is due to transmission line physics.

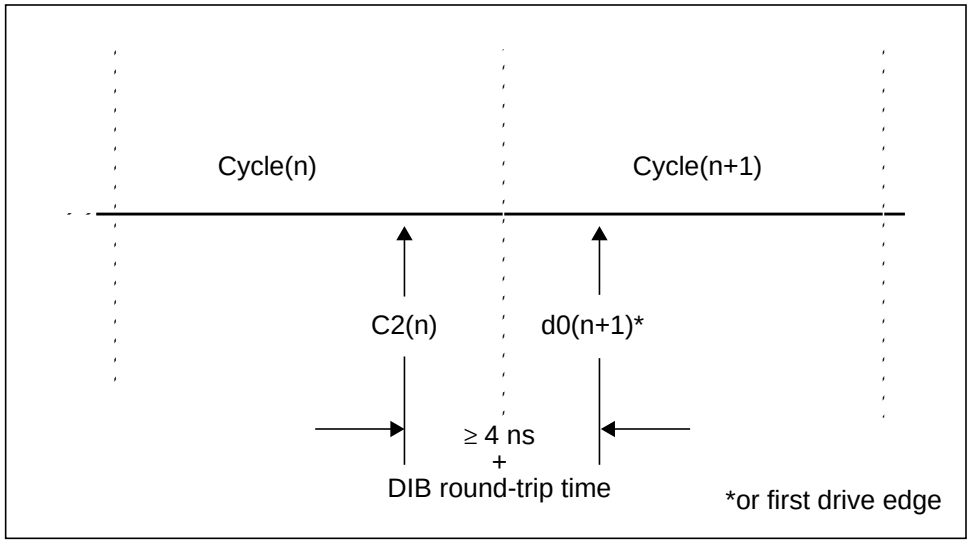


Figure 2.2-6. The Minimum Compare to Drive Time

7. For I/O channels in edge strobe mode, the c2 and c1 edges must be programmed to the same value.

8. For I/O channels in window strobe mode, the minimum window strobe width is 1.75ns. (For the J994, the strobe width is 3.0 ns.) Therefore, the minimum time between c1 and c2 is 1.75ns. In addition, the c2 of cycle(n) must be at least 6.5ns away from the c1 of cycle(n+1).

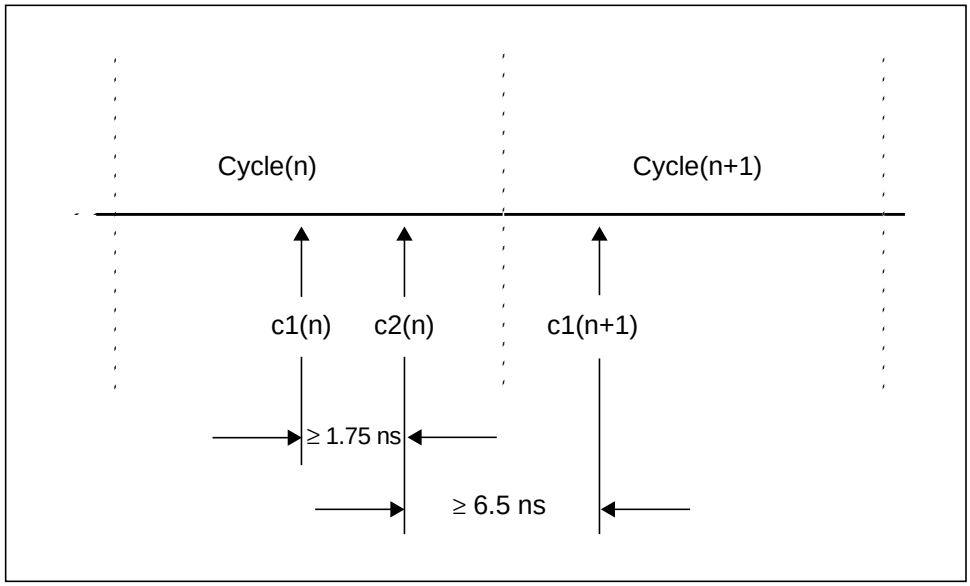
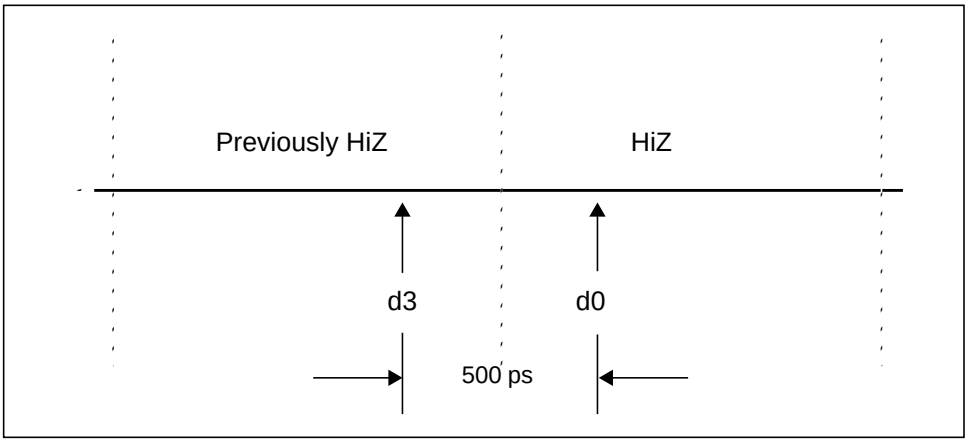


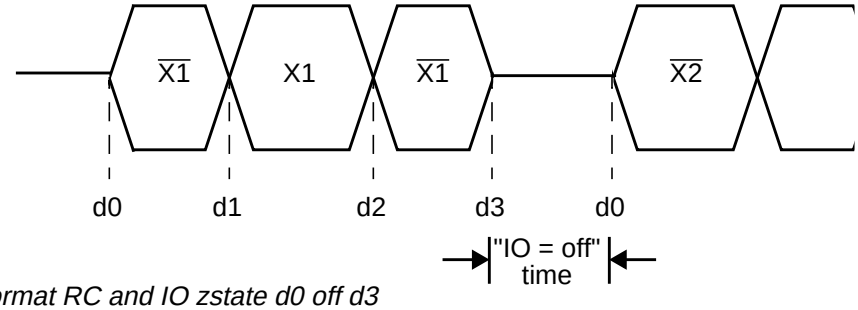
Figure 2.2-7. The Minimum Window Strobe Width

9. For IO channels in SCIO (d0 off d3) mode, if the cycle is to be a HiZ-only cycle (for "read-only" cycles), the d0 edge must be placed 500 ps after d3. Also, the previous cycle must end in a HiZ state. To achieve this "detect-only" cycle, the system should be specifically **calibrated** for SCIO with switching mode "d0 off d3".



10. For the IO switching modes "zstate d0 off d3" and "zstate d3 off d2", the minimum "IO = off" time is:

	chan-mux off	**chan-mux on**
100 MHz mode	3.5 ns	2.5 ns
60 MHz mode	7.0 ns	5.0 ns



Format RC and IO zstate d0 off d3

11. For the 60MHz pattern generator mode, the test system cycle time is between 16.984ns and 8.0µs. For the 100MHz pattern generator mode, test system cycle time is between 9.984ns and 8.0µs.

12. Edges can be delayed from one tester cycle into another as long as the following conditions are met.

    - ❑ The edge is not delayed for more than 8µs.
    - ❑ For the 60MHz pattern generator mode, the edge is not delayed for more than 4 cycles. Also, the edge is not within 17ns of the end of the 4th cycle (within 17ns of the end of the 3rd cycle for J993-01 systems).
    - ❑ For the 100MHz pattern generator mode, the edge is not delayed for more than 8 cycles. Also, the edge is not within 17ns of the end of the 8th cycle.
    - ❑ The edge does not cross a like edge.

13. In all formats, d1 must be less than d2. For drive-only channels, d4 must be less than d5.

14. When channel muxing, the time between the cycles must be ≥ 5 ns.

    For example, channel 200 is muxed with channel 201. These muxings take place on d3. The time between the d3 edge of channel 201 and the next d3 edge of channel 200 must be ≥ 5 ns.

    For drive-only channels, these restrictions are for the d0 edges.

15. For drive-only channels in formatter mux mode, if both the X and Y half of the channel are driven, the d3 to next d0 time must be at least 4ns.

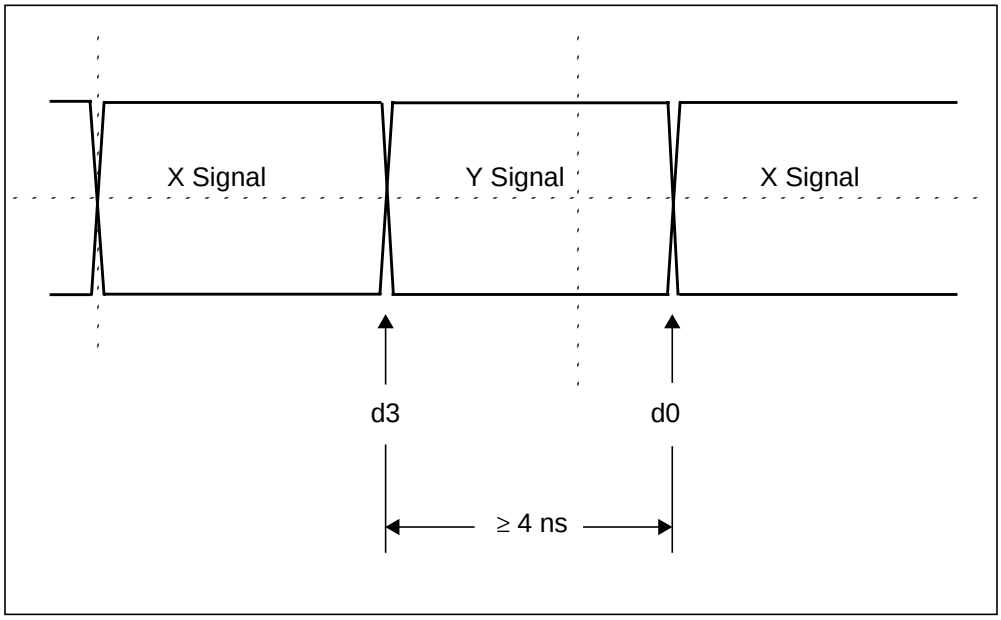


Figure 2.2-8. Minimum d3 to Next d0 Time

16. For drive-only channels in formatter mux mode, if the cycle is a Y only cycle (for page mode or similar mode), the d0 edge must be placed 500ps after d3. Also, the previous cycle must end with Y. Using the MCS hardware, X only cycles follow the same timing guidelines as the Y cycles.

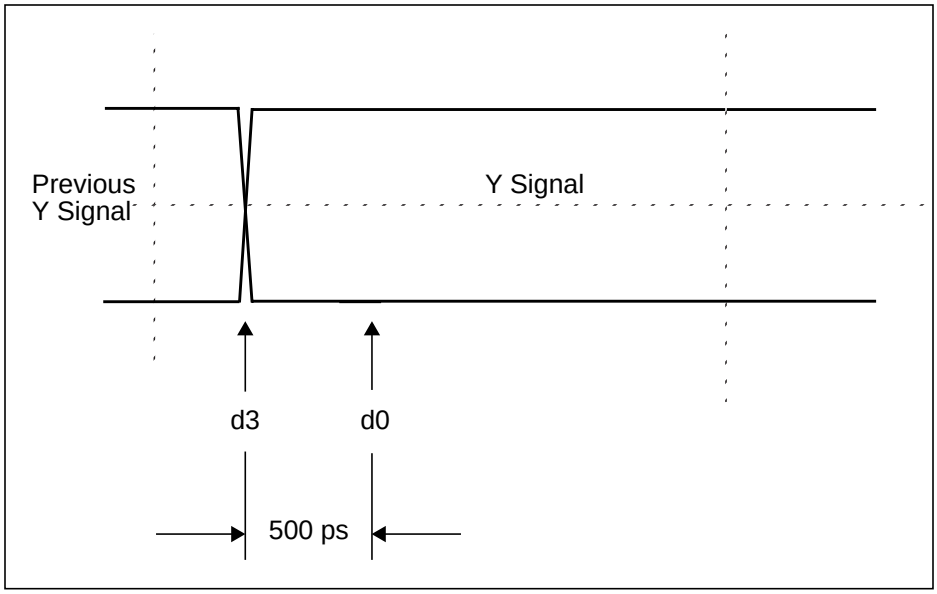


Figure 2.2-9. Y Only Cycle, Minimum d3 to Next d0 Time

17. When using NR (Non-Return) format on I/O channels or drive-only channels not in formatter mux mode, the d0 edge must be placed 2ns after the d1 edge. The transition will occur at the d1 edge.

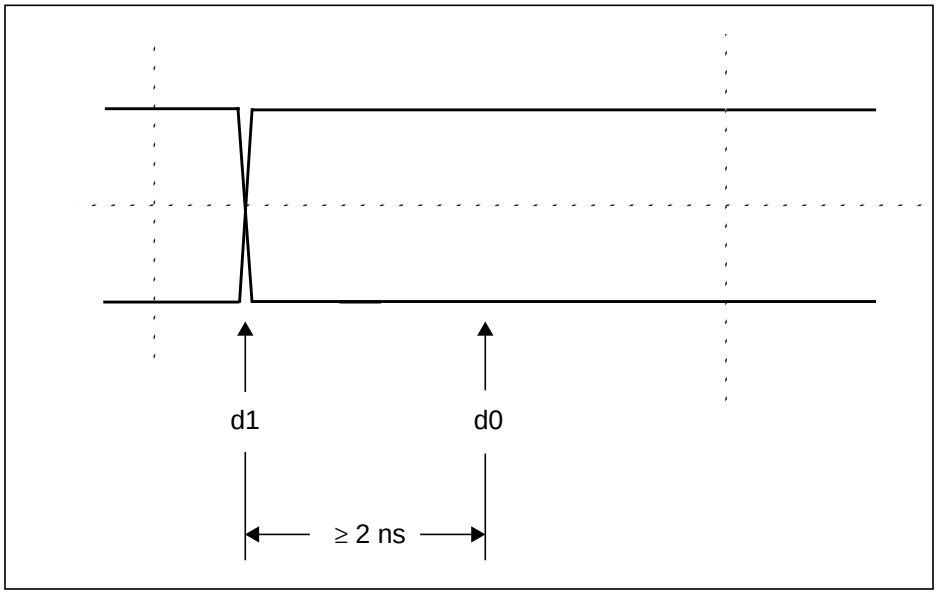


Figure 2.2-10. Minimum d1 to d0 time (NR Format/None Formatter Mux Mode))

18. When using NR (Non-Return) format on drive-only channels in formatter mux mode, for the X signal, the d0 edge must be placed 2ns after the d1 edge. The transition, however, will occur at the d0 edge. Likewise, for the Y signal, the d3 edge must be placed 2ns after the d4 edge. The transition, however, will occur at the d3 edge.

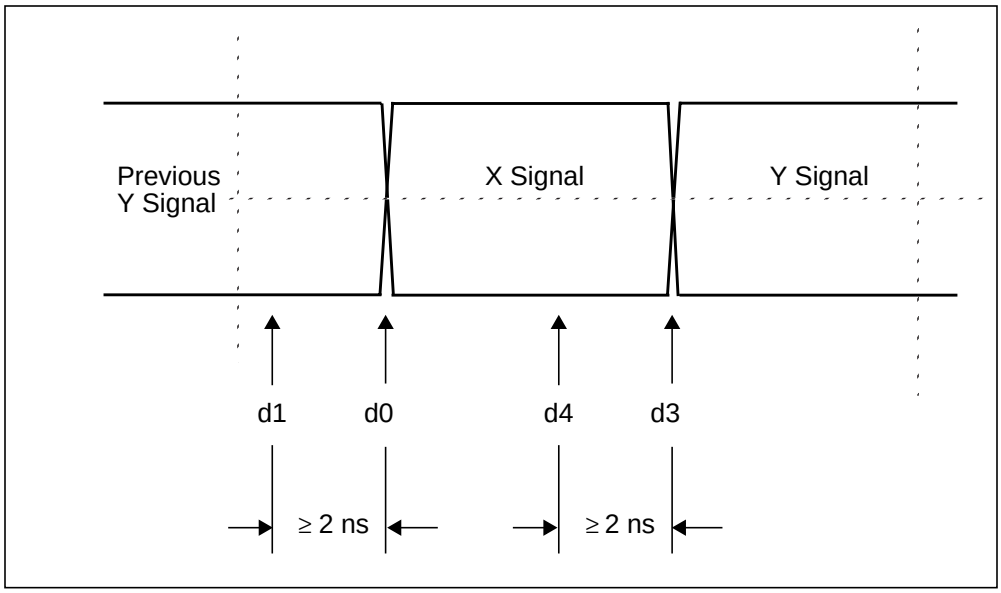


Figure 2.2-11. Minimum d1 to d0 and d4 to d3 Times (NR Format/Address Mux Mode)

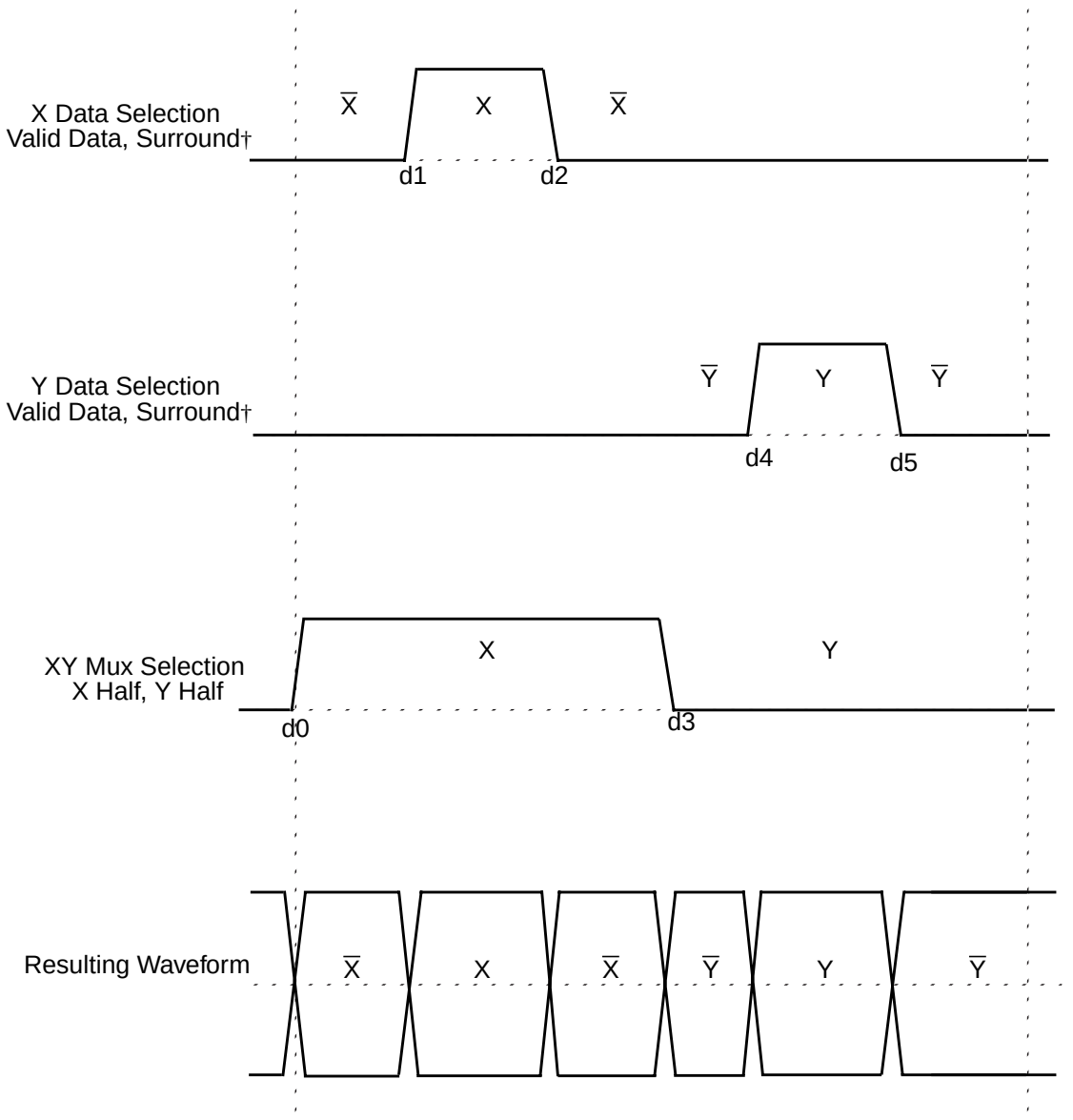
Drive-Only Timing Edges

For drive-only channels in address mux mode, the programmer has available six timing edges to setup the desired waveforms. Each of these edges has a specific purpose.

Table 2.2-2: Drive-Only Edges

Edge	Purpose
d0	This edge controls the format muxing for going to the X half of the channel. For channel muxing, this edge muxes the specified channel's waveform to the output. It is also used to set the X surround data for the cycle.
d1	This edge selects the valid data for the X portion of the cycle.
d2	This edge selects the surround data for the X portion of the cycle. Under mach2 mode, this edge drives the second portion of the X data.
d3	This edge controls the format mux for going to the Y half of the channel. It is also used to set the Y surround data for the cycle.
d4	This edge selects the valid data for the Y portion of the cycle.
d5	This edge selects the surround data for Y portion of the cycle. Under mach2 mode, this edge drives the second portion of the Y xdata.

The following figure illustrates the use of these edges:



† Note that the example shows *surround-by-complement* format (sometimes referred to as *return-to-complement* format). The surround data can also take other values. For example, in *surround-by-low format* (sometimes referred to as *return-to-low* format), the surround data is zero.

Figure 2.2-12. Drive-Only Timing Edges

Input/Output Timing Edges

For IO channels, the programmer has available six timing edges to setup the desired waveforms. Each of these edges has a specific purpose.

Table 2.2-3: Input/Output Edges

Edge	Purpose
d0	This edge selects the initial surround data. For "zstate d0", this edge controls the state of the drivers. For "zstate d0 off d3", this edge turns *on* the drivers.
d1	This edge selects the valid data for the cycle.
d2	This edge selects the final surround data for the cycle. Under mach2 mode, this edge drives the second piece of data. Under "zstate d3 off d2", it turns *off* the drivers.
d3	This edge controls channel muxing. For "zstate d3", it controls the state of the drivers. For "zstate d0 off d3", this edge turns *off* the drivers. and for "zstate d3 off d2", this edge turns *on* the drivers.
c1	This edge is the beginning of the window strobe.
c2	This edge is the end of the window strobe. Under edge strobe mode, c2 should be set equal to c1.

Guidelines For Using NR

The non-return format is used to drive data to the DUT without providing complement data transitions. For J990 test systems, there are specific issues that will help the programmer use the NR format effectively.

For I/O channels and drive-only channels not in formatter mux mode, the d0 edge should be programmed 2ns after the d1 edge. This allows the formatter to correctly set up the data for the cycle. The transition for the device will occur at the d1 edge. (See Figure 2.2-10.) Please note, d2 = (d1 + 2ns) unless otherwise specified.

For drive-only channels in address mux mode, the placement of the d0 and d1 edges is the same as above, but in addition the d3 edge for the Y half of the cycle should be placed 2ns after the d4 edge. Unlike I/O channels, however, the edge transition to the device will be seen at the d0 and d3 edges. (See Figure 2.2-11.)

Another issue arises when going from NR format in one cycle to a return format (Return High, Return Low, Return to Complement) in the following cycle. For the first cycle out of NR format, the initial complement or return data is not available.

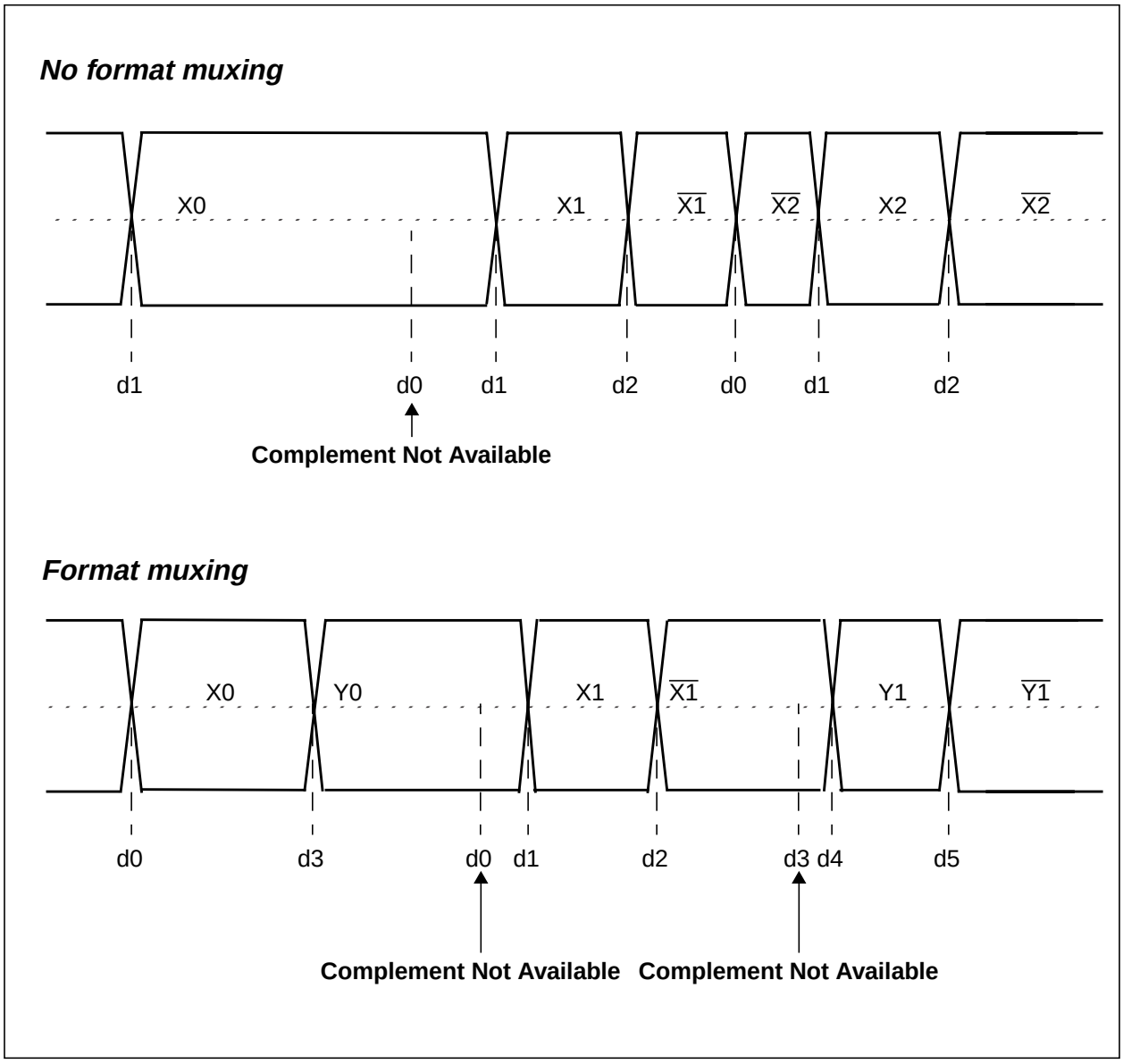


Figure 2.2-13. Initial Complement For NR format

If necessary, a *dummy cycle* can be used in the test pattern to allow the formatter time to reset so it can provide complement data.

Chapter 3.1: Using Templates

Introduction

After the databases for a test program has been developed, *templates* are used to generate instructions for specific tests. In general, you select the template for the test you want to perform and then input the data for the particular test into the template.

A template can be used over and over, and different sets of data may be applied to it. One set of data and the name of a template to which the data is applied is called an *instance*. An *instance database file* holds one or more instances. A job may use one or more instance database files.

The objective of this chapter is two-fold: to describe opens, shorts, functional, leakage, and ICC tests, and to show how to create an instance database file that contains instances for each test. Instances can be created either with an IG900+ tool (called **Insted**) or with a text editor and ADR syntax. For a discussion of how to use Insted see Chapter 1.11.

Basic Steps

The basic steps to creating instance database files are:

1. Decide what tests must be done
2. Decide which templates provide the test methodology to perform the test
3. Use ADR syntax or the IG900+ tool Insted to create the instances and instance database files

---

Instance database files do not have to be translated with the **adr900** command.

To run instances in a job, a workspace file must be created with the following in its *(group...)* statement:

- ❒ The instance database file name listed in the job clause of the group statement.

For more information regarding workspace and the (group...) statement, see Chapter 1.10.

In addition to this, the flow control file (main.t) must include calls to the instances that you want to run.

ADR Syntax

Instances are easily created using ADR syntax and an ASCII file. Either a text editor or the ADRedit tool can be used. The instance syntax is as follows:

```
(define-instance  INSTANCE_NAME  TEMPLATE_NAME
      (value   PARAMETER    DATA                    )
      (value   PARAMETER    DATA                    )
      (value   PARAMETER    DATA                    )
) ;END DEFINE-INSTANCE
```

where *define-instance* and *value* are reserved words, *PARAMETER* is the reference to the template field, and *DATA* is the field data that you will enter. The following is an example of a basic functional instance:

```
(define-instance basic_functional func_Template
        (value stop_on_first_fail 0 :choice “no”)
        (value patterns four_cycle_patt march_diagonal_patt
min_max_patt march_patt addrcompl_patt)
        (value dotests 1 :choice “yes”)
        (value autotitle 0 :choice “no”)
        (value tnum )
        (value trap )
        (value skip )
        (value driveroff )
        (value driver0 )
        (value driver1 )
        (value calset )
        (value waves wave_timing_nominal)
        (value levels nominal_levels)
        (value title )
        (value patstop )
        (value patstart )
        (value postbody )
```

```
        (value prebody )
        (value posttest )
        (value pretest )
) ;END OF BASIC_FUNCTIONAL
```

Instances use the following syntax rules:

- *define-instance, INSTANCE_NAME, TEMPLATE_NAME* and *value* must be used in the syntax and placed in the order shown in the above examples.
- The matching parenthesis structure must be followed.
- **Each instance must contain all of the referenced template's PARAMETERS, even if the instance does not require it. Parameters that are not programmed have no data after the parameter name.** The Insted tool creates the complete framework for you. For a list of each template's parameters, see the *J990 Series Template Reference Manual*.
- Some of the PARAMETERS require DATA, others do not. For a description of each parameter and its requirements, refer to the *J990 Series Template Reference Manual*.

More than one instance can be placed in an instance database file. Examples of this are provided in this chapter.

Procedure Overview

The remainder of this chapter describes various instances listed in the table below. Additional instances are provided in the demo program.

Table 3.1-1
Example Instances

Instance Name	Template Name	Job	Purpose
init_program_variables	background_Template	d1Mx4 DRAM	Background Setup
continuity	pin_pmu_Template	d1Mx4 DRAM	Continuity Test
input_leakage_ganged	pin_pmu_Template	d1Mx4 DRAM	Input Leakage Test
io_hi_leakage	pin_pmu_Template	d1Mx4 DRAM	Output Leakage Test
icc_ras_refresh	ivs_Template	d1Mx4 DRAM	Icc Test
icc_operating	ivs_Template	d1Mx4 DRAM	Icc Test
basic_functional	func_Template	d1Mx4 DRAM	Functional Test
funct_spec_levels	func_Template	d1Mx4 DRAM	Functional Test

Background Initialization Instances

The following instance from the d1Mx4 DRAM demo program deals with background initialization.

Background Initialization

The Background Template (background_Template) initializes many job level setup parameters at the beginning of the job program. (Do not confuse this with setting up for an individual test within a job execution; this is usually done with the test template for the test you wish to execute, i.e. functional test template or pin PMU template and so on.) To create a background instance, simply add an instance referencing the background_Template to any one of your job's instance database files (using Insted or a text editor) and reference the file in your flow.

As a rule, you should always execute an instance of the background_Template once per job, as the first instance executed in the job flow control file. This is true even if you don't plan to program any of the background parameters or change them from their default state. As you can see from the example below, this instance does not need to be complex. New instances of the Background Template may be executed at any time through the main.t file to change job level parameters.

Example Instance

```
(define-instance init_program_variables background_Template
        (value dotime 0 :choice “no”)
        (value title )
        (value autotitle 0 :choice “no”)
        (value disable_comp_pins )
        (value utilbits_clear )
        (value utilbits_set )
        (value floatpins )
        (value error )
        (value alarm )
        (value jobstop )
        (value patstop )
        (value patstart )
        (value posttest )
        (value pretest )
)
```

Notes

The example instance above shows how the background_Template is used. The first line defines the name of the instance.

```
(define-instance init_program_variables background_Template
```

Here the instance is called init_program_variables.

In this instance most template parameters are not used. All template parameters for the background_Template are optional, except two which have defaults (dotime and autotitle). This instance does not program any of the optional parameters; no data is provided for those parameters. The two values programmed are set to the same level as the defaults. It is not necessary to program defaulted parameters; however, it is very helpful for readability to specify the defaults explicitly.

The two values programmed are:

```
(value dotime 0 :choice “no”)
        .
        .
        .
(value autotitle 0 :choice “no”)
```

The **dotime** parameter specifies whether the instance execution times will be displayed. A 0 after the dotime keyword indicates no execution times will be displayed (the default). A 1 after the dotime keyword indicates an execution time will be displayed. The :choice “no” segment is for the Insted display. The Insted window will show the dotime parameter as *no*. This is more human readable than 0 or 1.

The **autotitle** parameter specifies whether the template name and instance name will be automatically printed in the datalog. In this example autotitle is set to 0, meaning no. This is the default.

No other parameter is programmed in this example.

Using the Background Template

To use this instance you would include it in your main.t program in the initialization section of your program, for example:

```
import Instance
        init_program_variables,
        fvm_initialization,
                .
                .
                .
```

Following this you would include the instance in your test flow. For example,

```
EXECUTE(init_program_variables);
```

The EXECUTE macro is used to execute an instance. The EXECUTE macro is described in the **AT900 Reference Manual**.

Continuity Tests—Background Information

This section provides a background discussion about continuity tests. The following section will provide an example instance along with an analysis of the instance.

Test Description

Continuity ("shorts and opens") tests are tests for shorted and open connections within the device — that is, for excessively low impedance paths ("shorts") and excessively high impedance paths ("opens") to adjacent device pins, power supplies, and ground.

In addition, shorts and opens tests identify electrical malfunctions in all parts of the test environment circuit — the device, the tester, the DIB, and the DIB-to-device interface (probe card, socket, and handler contactors).

For these reasons, test programs almost always include continuity tests at the beginning of the test flow. These tests not only identify "bad" devices; they screen out those circuits that have such gross defects that further testing would be a waste of time and money.

Continuity tests can be done using the functional hardware system or pin PMUs. Sometimes a program will execute a small number of functional tests, and if they pass, it is reasonable to assume that there are no opens or shorts. This can be a desirable approach because a small number of functional tests can be executed faster than PMU tests. If the functional tests fail, however, PMU tests would de required to determine which pins are shorted or opened. This is the approach used in the d1Mx4 demo program; it will be described in this section.

Most digital devices have their inputs and outputs protected by diodes connected to ground, as shown in Figure 3.1-1.

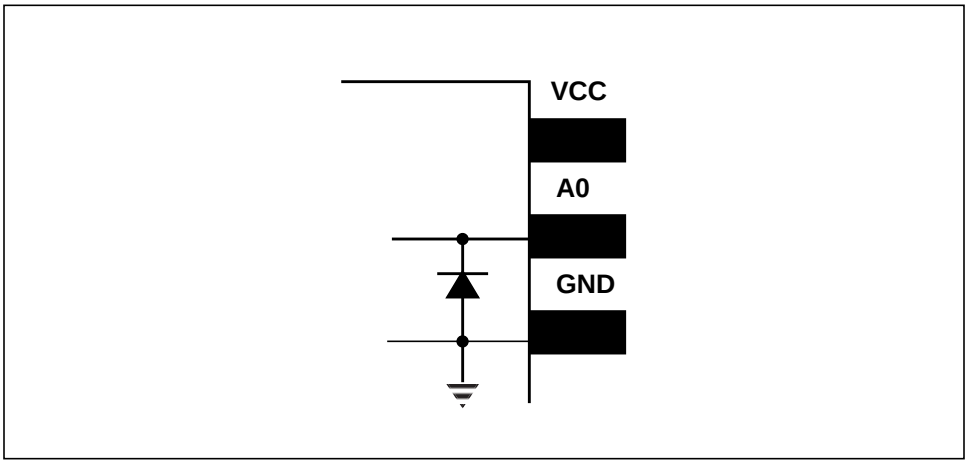


Figure 3.1-1Static Protection Diode to Ground

These diodes prevent static damage to the internal circuitry of the device.

This assumption is not valid for all devices, however, and care should be taken when adapting the sample program. A device with analog inputs, for example, may not have static protection diodes at all, and digital CMOS devices tend to have structures like that shown in Figure 3.1-2.

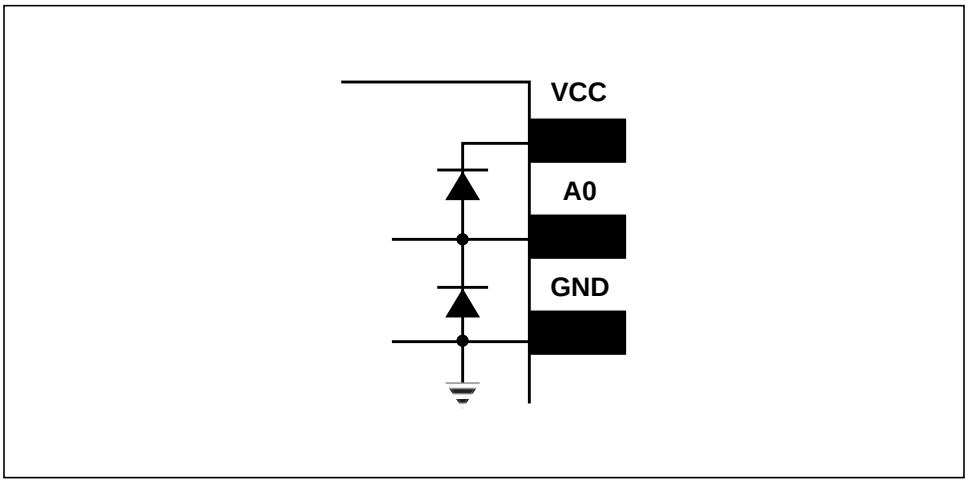


Figure 3.1-2 Static Protection Diodes to Power and Ground

Here, static protection diodes are connected to the power bus as well as to ground. In this case, shorts and opens tests may need to be done twice, reversing the voltage and current polarities to test the diode to the power bus.

PMU Shorts and Opens Test

The pin PMU is commonly used to test for continuity. In general, the method is to force a small negative current on one pin while all other pins are held at 0V. Voltage on shorted pins will be negligible; voltage on open pins will be high.

In general, to test for shorts and opens with the pin PMU, do the following:

1. Set all power (VCC) and ground pins to 0V.

   This is programmed using the levels parameter in the instance.

2. Using the pin PMUs, establish a background of 0V on all signal pins.
3. Set the pin PMUs to force a small negative current. Generally, a value of about –100µA is selected. Make sure the clamp voltage is appropriate for the memory-under-test.

   In the example instance, –100µA is used. The forceval1 parameter specifies the value.

4. Set test limits to an appropriate high and low level for the memory-under-test. A high limit failure indicates a *short*, and a low limit failure indicates an *open*.

   A typical value for these limits are -0.1 V and -1.0V

   This is programmed using the lolimval and hilimval parameters in the instance.

The appropriate template will be one of the pin PMU templates: *pin_pmu_Template*, *pin_pmu_table_Template, pin_pmu_alternate_Template, or pin_pmu_alternate_table_Template.*

The J994 test system also uses the pin_pmu_Template even though these systems have site PMUs. For the differences see the sequence discussion on page 3.1.10.

Continuity Test Instances

Below is an example instance as it appears in the d1Mx4 DRAM demo program. An instance such as this could be created using Insted or a text editor.

Example Instances

```
(define-instance continuity pin_pmu_Template
        (value forceval1 -0.0001)
        (value measure 0 :choice “Voltage”)
        (value irange 14 :choice “200UA”)
        (value lolimval -2)
        (value hilimval -0.2)
        (value pinlist DIODE_PINS)
        (value fload 0 :choice “off”)
        (value mpause 0.015)
        (value sequence 1 :choice “seq”)
        (value autotitle 0 :choice “no”)
        (value forceval2 )
        (value forcespec2 )
        (value tnum2 )
        (value forcespec1 )
        (value tnum1 10)
        (value lolimspec )
        (value hilimspec )
        (value stoplabel )
        (value startpat )
        (value floatpins )
        (value waitflagsfalse )
        (value waitflagstrue )
        (value duringpat )
        (value beforepat )
        (value driveroff )
        (value driver0 CLKS ADDRS DQS)
        (value driver1 )
        (value calset )
        (value waves )
        (value levels power_off_levels)
        (value title2 )
        (value title )
        (value patstop )
        (value patstart )
        (value postbody postbody_function )
        (value prebody prebody_function)
        (value posttest )
        (value pretest )
)
```

Notes

Continuity tests are implemented with the pin_pmu_Template in this instance. This instance of the pin_pmu_Template is simply called continuity:

```
(define-instance continuity pin_pmu_Template
```

The first few values in the instance set the PMU voltages:

```
(value forceval1 -0.0001)
(value measure 0 :choice “Voltage”)
(value irange 14 :choice “200UA”)
(value lolimval -2)
(value hilimval -0.2)
```

The **forceval1** parameter is set to -0.0001. Forceval1 is a double-precision floating point number. It is the value to be forced by the PMU. Here the force value is a current, this is because the measure value is a voltage. The PMU will force a voltage and measure a current of vice-versa. Volts or amperes are the presumed units. For example 10 mA must be entered as 10.0e-3 or 0.01.

Either forceval1 or **forcespec1** must be programmed. For the forcespec1 parameter, you name a spec from the levels database, and that spec will be the PMU force value. If both the forceval1 parameter and the forcespec1 parameter is specified, the forcespec1 parameter will be used. **Forceval2** and **forcespec2** are used if the PMU will be used for a second set of tests. The **tnum1** parameter (test number to be assigned the in conjunction with forceval1) is used for the first set of tests, and **tnum2** parameter will be used with the second set of tests. In this example forceval2, forcespec2, and tnum2 are not used.

The **irange** (current range) parameter is a choice of:

13 (2MA)
14 (200UA)
15 (20UA)
16 (2UA)
17 (200NA)

Default = 16 (2UA).

All five ranges are available for sensing current, but only ranges 13 (2MA) and 14 (200UA) are available for forcing current.

The **lolimval** and the **hilimval** parameters specify the limits the constitute pass or fail for the measure parameter. In this case voltage is being measured. A value between -2 volts and -0.2 volts will constitute a pass. Values outside this constitute a fail. Although the demo program uses this range, the following typical range is more common: -1 volts (lolimval) to -0.1 volts (hilimval).

```
(value pinlist DIODE_PINS)
(value fload 0 :choice “off”)
(value mpause 0.015)
(value sequence 1 :choice “seq”)
```

The **pinlist** parameter lists one or more pins on which tests are to be run. The PMU tests will be run on all pins indicated. If two forceval parameters are specified, both will run on all pins listed. The pinlist is defined in the pindb. In this example, DIODE_PINS are defined as (DQS CLKS ADDRS).

The **fload** parameter, which is turned off in this example, allows the dynamic current load (ILOAD) circuitry to be used during testing to force current on the pins. (Only IO channels have ILOAD circuitry.) The ILOADS can provide forced current

greater in magnitude than the pin PMUs can provide. The ILOAD current sources are capable of forcing as much as 50 mA. For pins that will use this feature, the pins must be in HiZ (compare mode) — use the driveroff parameter to ensure this.

The **mpause** parameter is used to allow PMU settling. This pause will allows the force value to stabilize. The mpause values specified is a double precision floating point number. It ranges from 0.0 to 17.0e-3 (17 ms). The default value is 15.0e-3 (15 ms). The specified value in this instance is the same as the default. This actual value has been specified in the instance as a memory aid. The demo uses the default value, but it may be quite safe to use a smaller value. The mpause in this example could probably go down as low as 0.005 ms.

The **sequence** selection is a choice between 0 (ganged) or 1 (sequential). This selection determines if the results will be ganged or individual. If the results are set to ganged, all the results of tests are ORed, meaning that a fail on any pin in the pinlist will result in a fail in the datalog. Sequential means that the results of each pin pmus results will be listed separately in the datalog. For the J994, which uses site PMUs instead of pin PMUs, a sequential result selection here means that the site PMU will test each pin at each site separately and datalog individual pins. Ganged on the J994 is the same as on other systems with all test results ORed together. In this example the sequence parameter is set to sequential for individual results.

For an explanation of autotitle see page 3.1.5.

```
(value driver0 CLKS ADDRS DQS)
                .
                .
                .
(value levels power_off_levels)
```

The **driver0**, **driver1** and **driveroff** parameters all take a pinlist. The driver0 parameter sets the drivers in the specified pinlist to VIL. The driver1 parameter sets the drivers in the specified pinlist to VIH. the driveroff parameter turns the specified pinlist drivers off (HiZ). The VIL and VIH values are provided by the levels specified using the levels parameter (described below). In this example the clocks, address, and DQ pins are programmed using the driver0 parameter, so they are set to the VIL value.

To avoid confusion, each pin should be programmed for only one of these driver conditions. The execution order is driver1, driver0, and driveroff. If the same pin appears in more than one of the driver parameters, the last state specified will be the state of the driver.

The **levels** parameter specifies the levels that will be used with the drivers and other DC values. This includes setting all power and ground pins to 0 in this instance. The level object is specified in the wspadr file inside a group. For example, in demo program the power_off_levels parameter references the following group clause:

```
        (group
                (default :comment nil)
                (group-form
                        (:type          :name   :elements)
                                        .
                                        .
                                        .
                        (level          power_off_levels
                                                (
                                                power_off_levs
                                                levels_eqns
                                                )
                        )
```

The power_off_levs values can be found in the specadr, and levels_eqns are defined in the leveladr.

The last two parameters programmed in the instance are the **postbody** and **prebody** functions:

```
                (value postbody postbody_function )
                (value prebody prebody_function)
```

The prebody parameter allows a function to be executed prior to executing the instance, and the postbody parameter allows a function to be executed after the instance is executed. These are usually used for preconditioning and post processing of results. The prebody_function and postbody_function for the demo program is located in the support_modules.t file. This prebody_function starts a timer to time how long it takes the instance to execute. The postbody_function stops the timer and prints out the amount of time it took to execute the instance.

Using The Continuity Instance

To use this instance import it into your main.t file. For example,

```
 import Instance
                init_program_variables,
                        .
                        .
                        .
                continuity,
                        .
                        .
                        .
```

Next, include the instance in your test flow. For example,

```
EXECUTE(basic_functional);
FAIL_DO(basic_functional) {
        EXECUTE(continuity);
        FAIL_DO(continuity) {
                sort_bin(FAIL_CONTIN);
        }
        PASS_DO(continuity) {
                sort_bin(FAIL_BFUNC);
        }
}
```

This section of the main.t program checks continuity with both the functional test system hardware and the PMU. The first line executes an instance called *basic_functional*. The basic_functional instance is discussed in the next section of this chapter. This instance runs a simple march diagonal pattern. If the functional tests complete without failure, no continuity tests will be executed.

The FAIL_DO macro begins a loop which will execute the continuity template. Note that FAIL_DO and PASS_DO are two other macros often used in conjunction with EXECUTE to apply instances. FAIL_DO and PASS_DO execute blocks of code that only affect devices that have failed or passed (respectively) the executed instance. EXECTUTE, FAIL_DO, and PASS_DO are documented in the **AT900 Reference Manual**.

The sort_bin statements are function calls to another routine that controls binning. The DUT will be binned to either the failed continuity bin or the failed basic functional bin.

Functional Tests—Background Information

Background about functional testing is provided first in this section, followed by instance examples and analysis in the next section. Functional tests are among the most important tests in a test program, and some patterns can be very complex. Fortunately, the functional test instance is very straightforward. It is almost entirely a matter of selecting a pattern to use from the pattern databases, a wave from the wave database, and a level from the levels database. This is essentially all you do at the instance level. (Of course these databases need to be created with the right data in them. The creation of these databases is covered in the *Writing the Test Program* section.)

The background discussion reviews the considerations involved in functional testing to help guide in the creation of appropriate functional instances. If more details are desired about any of the test system hardware or formats refer to the *Programmer's Guide to the Hardware* section.

Test Description

Functional tests verify that the device operates as described in the manufacturer's data sheet. To perform a functional test, patterns of low and high voltages (interpreted as logical zeros and ones) are placed on device input pins, and the subsequent voltages on device output pins are measured (and reinterpreted as ones and zeros).

If the device is functioning properly, each input pattern should produce a given, predictable output pattern when the memory-under-test is read back. By using a correctly designed series of input patterns, every cell of the device can be verified as operating correctly.

In theory, a very large number of different and useful functional tests could be performed on a memory device. Time and cost, however, require that functional tests be as efficient as possible. Balancing these objectives can be a challenging part of test engineering design and should take into account the probable failure modes of the device and the consequences of not detecting a failure versus the economics of more testing.

Functional System

Let's examine the basic components that make up the functional system. To perform a functional test on a device, the tester must provide:

- ❒ A voltage source to power the device
- ❒ An input system to propagate patterns at the proper voltage levels
- ❒ An mechanism to test device output, detect errors, and process failures

Functional testing resources are shown in Figure 3.1-3. Knowledge of these resources is required before functional tests can be written. Each is discussed in the section that follows.

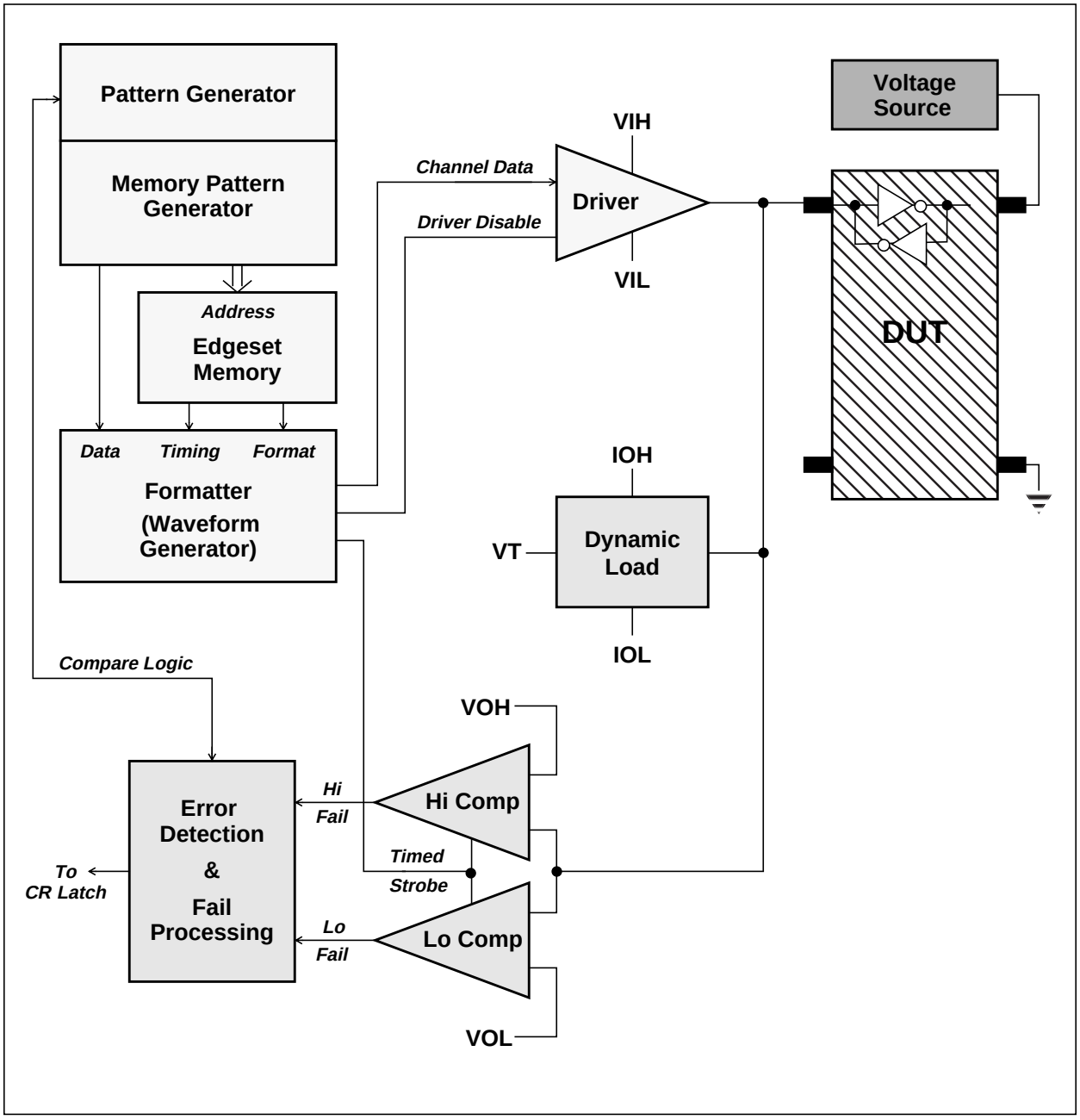


Figure 3.1-3 System Basic Functional System

Voltage Source

The J990 Series Memory Test Systems vary in the number of voltage sources they have. The Square headed (handler) systems have 32 voltage sources and the Round headed (prober) systems have 16 voltage sources.

Voltage source connections are hardwired on the device interface board (DIB) with force and sense lines tied as closely as possible to the power pin(s) to provide a Kelvin connection.

Each voltage source has a programmable current limit.

Input Data and Levels

The system hardware must deliver waveforms to input pins with the precise waveshape, voltage levels, and transition timing edges specified by the device data sheet.

Waveforms are usually divided into the **data ("logic state") component**, and the **non-data ("edgeset") component**. Data components generally fall in the middle of the waveform, between two non-data components, and occur once per tester-cycle. Thus, the fastest speed a tester can attain is its minimum period. See the specifications for your system to determine the minimum period.

In a test system, information needed to create waveforms is stored as three separate elements: pattern information, format information, and waveform edge timing information.

This scheme makes use of the following input hardware:

- ❒ Pattern Memory
- ❒ Pattern generator
- ❒ Timing system
- ❒ Formatting system
- ❒ Input device driver

Pattern Memory

Pattern memory contains the device truth table data, the patterns of ones and zeros by which the device will be tested. The pattern has input and output code for each channel of the tester. (Input data is sent in waveforms to input pins, and output or "compare" data is used as predicted data against which the actual values on the output pins will be compared.)

The pattern memory contains the data by which a logic 0 or 1 for each signal pin of the device is created, as well as edgeset addresses (containing code for applying the correct timing and format to logic data), ignore bits to enable and disable the comparators, and pattern sequence control codes ("opcodes") for controlling pattern generator steps.

Pattern Generator

The pattern generator ("patgen") provides data pattern sequence control. It executes locations in the pattern memory, either sequentially or under patgen memory opcode control. (Opcodes include such commands as stop, goto, call, return, and the like.)

The *writesys(patgen_start)* statement starts the pattern generator.

The pattern generator runs until a *stop* opcode is encountered in the pattern code, or a stop condition specified in the background template is encountered.

By default, the test system CPU pauses in program execution until the pattern generator has stopped running. In addition, the pattern generator will stop on first fail. You can override both of these defaults, however. For more information regarding this, please see the *AT900 Reference Manual* for documentation concerning the writesys(patgen_start, ...), writesys(patgen_stop_mode, ...), and *writesys(patgen_stop)* statements, as well as the background conditions template.

Timing System

The timing system places edge transitions and timed strobe points at programmable locations in the waveform.

Programmable edges for timing purposes are

t0, a tester reference point for each pin from which edge transitions are measured. One **period** is the time from any t0 to the next t0.

D0, the leading edge in a waveform transition to form the surround data for a return type format. It does not need to be the same as t0.

D1, the **leading edge** in a waveform transition. The D1 edge causes the transition to data from the surrounding format.

D2, the **trailing edge** in a waveform transition. The D2 edge causes the transition from data back to the surrounding format.

D3, the **MUX** or **I/O switching edge**. D3 is used to control format multiplexing, channel multiplexing, and I/O switching

D4, the **leading edge** in a waveform transition for the second piece of data for cycle. The D3 edge cause the transition to data from the surrounding format.

D5, the **trailing edge** in a waveform transition for the second piece of data for a cycle. The D5 edge cause the transition from data back to the surrounding format.

C1, the leading edge in an output strobe. The **C**1 edge causes the compare window to open.

C2, the trailing edge in an output strobe. The **C2** edge causes the compare window to close.

Timing information is stored along with format information in a memory called *edgeset memory.* Edgeset memory can store up to 32 different timing and format combinations, all of which are available for a given pattern burst.

Formatting System

The formatting system allows you to define the shape of a waveform by determining the behavior of the part of the wave that is not vector data.

The RL ("return low") format, for example, forces the device input voltage level low (to VIL) in the non-data areas of the waveform. Another format, RH ("return high"), forces non-data voltage levels high (to VIH). Combined format and timing information for each pin is stored in edgeset memory.

Pin Electronics Ranges

There are three pin electronics ranges for J990 Series systems, for all channel resources including the pin PMUs. The ranges can be programmed on a per test head basis:

-4v to +5v
-3v to +6v
-2v to +7v

Since it takes 750 ms to alter the levels range, changing ranges too frequently or using different ranges throughout your leveladr file will slow down your program.

Input Device Driver

J990 Series drivers can force voltages associated with the data as defined by the selected waveform. Each driver can put out a programmed input high voltage (VIH) corresponding to data 1 and an input low voltage (VIL) corresponding to data 0 to the device.

Output Testing and Fail Processing

As input voltages are applied to the input pins, the voltages on the output pins must be read and the results compared with the expected (predicted) values. The mechanism used to compare the actual device output with the expected output consists of the following hardware:

- ❒ Pattern memory
- ❒ Pattern generator
- ❒ Timing system
- ❒ Formatting system
- ❒ Comparator logic
- ❒ Dynamic current load
- ❒ Error detector and fail processor

With this hardware, the J990 Series test systems can strobe device output voltage levels, detect pin failures, and process failure information.

Pattern Memory, Pattern Generator, Timing and Formatting Systems

The pattern generator executes patterns from the pattern memory, selecting the appropriate edgesets (timing and format) for each cycle. A timed strobe of the device output is then compared with the expected output of the device. The result of this comparison is sent to the error detector and fail processor for use in fail detection.

Comparator Logic

The comparator logic (or *compare logic*) circuits compare the device output voltage level with the expected values. Two voltage values can be specified for each output pin, an output high voltage (VOH) and an output low voltage (VOL).

Comparators are programmed to look for an output value for a specified duration (*window strobe* mode) or at a specified time (*edge strobe* mode). Figure 3.1-4 shows a window strobe compare waveform.

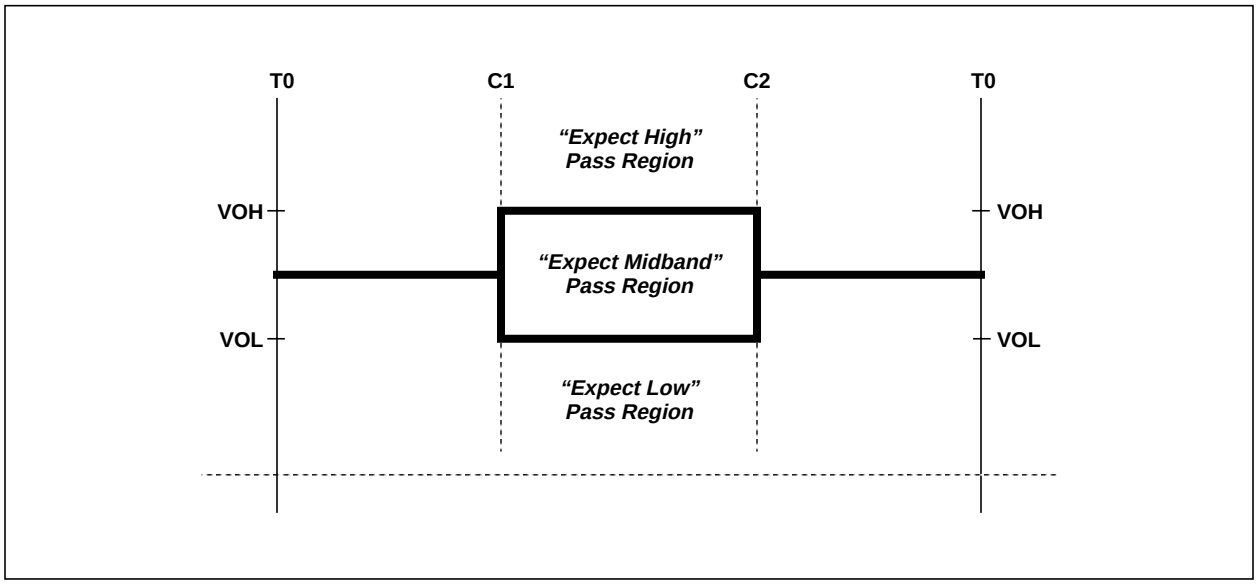


Figure 3.1-4 Compare Waveform (Window Strobe)

The comparator is programmed to look for a *high fail*, *low fail*, or *midband fail*. In window strobe mode:

If the expected data for the window is logic one, the comparator will expect the output for that pin to be greater than VOH for the entire time C1 to C2 (the *high pass* region in Figure 3.1-4). If the output level drops below VOH, a *high fail* will be processed by the error detection logic, and the cumulative reject (CR) latch will be set.

If the expected data is logic zero, the output must be less than VOL for the entire time C1 to C2 (the *low pass* region in Figure 3.1-4). If the output level rises above VOL, a *low fail* will be processed by the error detection logic, and the cumulative reject (CR) latch will be set.

If the expected data is a midband level, the output during the period C1 to C2 must be less than VOH and greater than VOL (the *midband pass* region in Figure 3.1-4). If the output level rises above VOH or drops below VOL, a *midband fail* will be processed by the error detection logic, and the cumulative reject (CR) latch will be set.

Edge strobe mode is similar to window strobe mode, except that data must be true only at C2 (C1 and C2 must be programmed the same).

VOH and VOL are programmable to any value within the pin electronics ranges, with a resolution of 0.5mV. The minimum width of a window strobe is 1.75 ns. The edge strobe is a gate transition and has no pulse width.

Dynamic Current Load

The dynamic current load circuit (ILOAD) is connected to the device output.

Dynamic load circuits simulate actual in-circuit loading conditions and allow device output pins to sink or source a measurable load current, depending on whether the output voltage is above or below a programmed threshold voltage, VT.

To program the dynamic current load for a pin:

1. Program VT to a point between the expected VOH and VOL levels.
2. Program IOH and IOL according to the device spec sheet.

During functional testing, the voltage on output pins swings toward either VOH or VOL. With ILOADs enabled, a voltage swing above VT (toward VOH) causes the device to sink current (from IOH), and a swing below VT (toward VOL) causes the device to source current (to IOL).

Dynamic load circuitry can also be used during continuity testing in place of the PMUs.

Functional Test Instances

This section describes functional test instances. The demo programs contain many other instances of the func_Template.

Instance Examples

```
(define-instance basic_functional func_Template
        (value stop_on_first_fail 0 :choice “no”)
        (value patterns march_diagonal_patt)
        (value dotests 1 :choice “yes”)
        (value autotitle 0 :choice “no”)
        (value tnum 1)
        (value trap )
        (value skip )
        (value driveroff )
        (value driver0 )
        (value driver1 )
        (value calset nominal_level_cal)
        (value waves wave_timing_nominal)
        (value levels nominal_levels)
        (value title )
        (value patstop )
        (value patstart )
        (value postbody postbody_function )
        (value prebody prebody_function)
        (value posttest )
        (value pretest )
)


(define-instance funct_spec_levels func_Template
        (value patterns march_patt addrcompl_patt)
        (value stop_on_first_fail 0 :choice “no”)
        (value dotests 1 :choice “yes”)
        (value autotitle 0 :choice “no”)
        (value tnum )
        (value trap )
        (value skip )
        (value driveroff )
        (value driver0 )
        (value driver1 )
        (value calset )
        (value waves wave_timing_nominal)
        (value levels spec_levels)
        (value title )
        (value patstop )
        (value patstart )
        (value postbody )
        (value prebody )
        (value posttest )
        (value pretest )
)    ; END OF FUNCT_SPEC_LEVELS
```

Notes

Functional tests are implemented with instances of the func_Template. Two sample instances of many are shown above: a basic functional test instance, and an instance that runs functional test with nominal spec levels. The demo program contains several other functional test instances for your use.

```
(value stop_on_first_fail 0 :choice “no”)
(value patterns march_diagonal_patt)
(value dotests 1 :choice “yes”)
```

The **stop_on_first_fail** parameter is a choice of yes or no. If the template contains more than one pattern, this parameter will determine if instance execution should stop after the first pattern failure.

The **pattern** parameter is used to specify the pattern name. It is a pattern as specified in the group clause of the workspace adr file (wspadr). Here the instance march_diagnal_patt is the selected pattern. This pattern is defined as:

```
(pattern        march_diagonal_patt
                        (
                        march_diagonal   ;startset
                        dram_patterns    ;mpgset
                        pattern_data     ;dgenset
                        srcset           ;srcset
                        d1Mx4_mut        ;mutset
                        )
)
```

One pattern or a list of patterns can be specified with this parameter. The second instance shown in the example above (funct_spec_levels) uses two patterns: a march pattern, and an address-complement pattern.

The **dotests** parameter specifies whether writesys(test_func) should be issued after a pattern is executed this will determine if functional pass/fail results are to be read back. The choices are 0 (no), 1 (yes), or 2 (fails only). In the example instances the choice of yes is selected. This is the default.

```
(value calset nominal_level_cal)
(value waves wave_timing_nominal)
(value levels nominal_levels)
```

The **calset** parameter is used to select a calset for the instance. The calset will optimize calibration of the test system resources to the specific requirements of the instance. Creation of calibration database files (caladr, caldb) is described in Chapter 1.9 (*Writing the Test Program*). If a calset is not specified, calibration values are inherited from the context in which the instance is executed. For example, the main.t file may have already specified a calset to use using the writesys(select_cal_data) statement.

Once a calibration file has been created, it must be included in the workspace (wspadr) file, for example:

```
(job            j993_1Mx4
                        (
                        explicit_timing         ; waveform
                        wave_timing_70ns        ; waveform
                        wave_timing_80ns        ; waveform
                              .
                              .
                              .
                        "d1Mx4_calsets.caldb"   ; calsets
                              .
                              .
                              .
                        )
)
```

The **waves** parameter allows a waveform from the workspace to be specified. In this instance the wave_timing_nominal waveform is selected. This is defined in the workspace file as:

```
(waveform       wave_timing_nominal
                      (
                      timing_nominal
                      spec_wave_eqns_1
                      )
)
```

This segment is part of the group clause in the workspace file. It allows a waveform which consists of a timing spec object and a timing equation object. The timing_nominal element is defined in the d1Mx4_timing.specadr file. The timing_nominal element contains the nominal timing specifications. The spec_wave_eqns_1 is an equation set (eqset) from the wave database (d1Mx4_wave_eqns.waveadr). It contains the timing equations and formats.

The **levels** parameter has been previously described (see page 3.1.10). The remaining specified parameters (**prebody** and **postbody**) have also been described (see page 3.1.11).

Using Functional Test Instances

Functional instances are used just as other instances are: you use the **import Instance** statement to import the instance into your main.t file and then include the instance in your test flow. For example the func_spec_levels instance is used as follows in the demo program:

```
EXECUTE(funct_spec_levels);
FAIL_DO(funct_spec_levels) {
      sort_bin(FAIL_FUNCT_LEVELS);
}
```

A more complex example that uses several functional test instances in a test flow is the following speed sort:

```
EXECUTE(funct_spec_70);
FAIL_DO(funct_spec_70) {
        EXECUTE(funct_spec_80);
        FAIL_DO(funct_spec_80) {
             EXECUTE(funct_spec_100);
             FAIL_DO(funct_spec_100) {
                      sort_bin(FAIL_FUNCT_SPEC);
             }
             PASS_DO(funct_spec_100) {
                      sort_bin(PASS_SPEC_100);
             }
     }
     PASS_DO(funct_spec_80) {
             sort_bin(PASS_SPEC_80);
     }
}
PASS_DO(funct_spec_70) {
     sort_bin(PASS_SPEC_70);
}
```

This speed sort portion of the main.t file will execute three functional test instances that test a various speeds: funct_spec_70, funct_spec_80, and funct_spec_100. The funct_spec_70 instance uses 70 ns timing; the funct_spec_80 instance uses 80 ns timing; and funct_spec_100 uses 100 ns timing.

Leakage Tests—Background Information

This section describes leakage tests.

Test Description

Leakage tests look at the current flowing through the device at the input and output pins. Because voltages are present at these pins under normal operating conditions, a certain amount of current leakage will occur. Current outside specified ranges, however, is considered excessive and indicates a defective device.

J990 Series test systems can measure and test for input and output current leakage, as well as "high-impedance" or "tri-state" output current leakage at the device pins.

Input current leakage is current flow into the reverse-biased diodes and input protection network connected to the gates in the device. Typically, the impedance through undamaged gates is high, and input current values are correspondingly low, often in the picoamp range. A defective gate or input protection network, however, can cause excessive current to flow. Input leakage tests look for excessive current through device input pins and I/O pins in input mode.

Output current leakage is current flow through the diodes and output protection network, if these exist. Output leakage tests look for excessive current through output-only pins.

"High-impedance" or "tri-state" output current leakage is current flow through output pins that have been placed in high-impedance mode. (High-impedance mode is sometimes called "hi-Z", "tri-state", or "off-state" mode.)

High-impedance mode is an intermediate state between "on" and "off". Current flow in this state should be minimal, since the presence of current, either sourced or drained, is an indication of output mode. High-impedance output leakage tests look for excessive current through I/O and output pins in high-impedance mode.

For all leakage tests, either the system PMU or the pin PMU can be used to force voltages on pins equal to the minimum and maximum input voltage levels. At these voltages, worst-case leakage will occur.

Current measured by the PMU at minimum and maximum voltages should be within the ranges specified in the device data sheet. (For devices that must be tested for output leakage or hi-Z output leakage, input current values should be used unless separate output and hi-Z output current values are specified by the manufacturer's data sheet.)

System PMU Resources

The DC measurement system provides both a system PMU and a pin PMU capability, either of which could be used to perform the input and hi-Z leakage tests.

Because of its greater speed, accuracy, and true parallel measurement capability, however, the pin PMU is recommended over the system PMU where possible. Let's look at these resources one at a time, starting with the system PMU.

System PMU

The system provides general parametric capabilities (the ability to force voltage and measure current, or force current and measure voltage) through a system or central PMU.

The system PMU can force and measure voltages within the following ranges:

System PMU
Voltage Force and Measure Ranges

±2 V
±20 V
±40 V

The system PMU can force and measure currents within the following ranges:

System PMU
Current Force and Measure Ranges

±2 µA
±20 µA
±200 µA

±2 mA
±20 mA
±200 mA

The force and sense lines of the system PMU can be switched to any channel of the test system through the FET matrix. The voltage and current ranges available through the FET matrix are limited, and the measurements are slightly less accurate. (See your respective J990 Series test system specification for more information on the system PMU.)

Note that while the system PMU could be used to test for leakage, it is a shared resource, and the test would have to be performed one pin at a time. Pin-by-pin testing is time-consuming, not only because of the increase in test time, but also because of the added per-pin connection and disconnection times.

System Pin PMU

In addition to a system PMU, the DC measurement system also provides pin PMU resources. Pin PMUs are specifically designed for small current measurements like leakage testing, and they have both accuracy and the speed advantages over the system PMU when used for this purpose.

The pin PMU can force current/measure voltages within the following ranges:

Pin PMU
Force Current/Measure Voltage Ranges

±200 uA
±2 mA
±50 mA

The pin PMU can force voltage/measure currents within the following ranges:

Pin PMU
Current Force/Measure Ranges

±200 nA
±2 µA
±20 µA
±200 uA
±2 mA

Program Pin PMU Leakage Tests

Programming a test system for a DC-type tests is a simple procedure. Note that by default the signal pins of the device are connected to the functional electronics (driver, comparator, and dynamic loads), and the power pin is connected to the voltage source (see Figure 3.1-5).

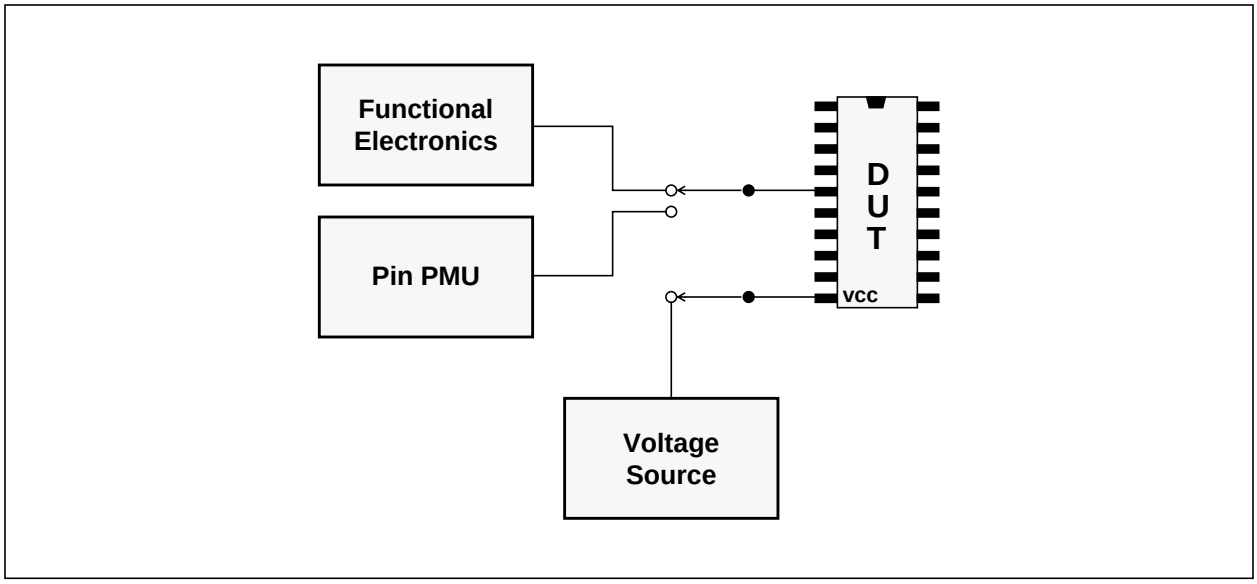


Figure 3.1-5 System Hardware Connections Prior to DC Testing

In general, to perform a DC measurement on a signal pin:

1. Program the pin PMU force and measure conditions.
2. Connect the pin to the pin PMU.
3. Perform the test.
4. Reconnect the pin to the functional electronics.

Leakage Instances

The following are example instances as they appear in ADR syntax. These examples show input and output leakage tests.

Instance Example

```
(define-instance input_leakage_ganged pin_pmu_Template
        (value sequence 0 :choice “ganged”)
        (value lolimval )
        (value hilimval )
        (value levels leakage_levels)
        (value lolimspec IL_lo)
        (value hilimspec IL_hi)
        (value fload 0 :choice “off”)
        (value mpause 0.015)
        (value irange 15 :choice “20UA”)
        (value measure 1 :choice “Current”)
        (value autotitle 0 :choice “no”)
        (value forceval2 5.5)
        (value forceval1 0)
        (value pinlist INPUTS)
        (value pretest )
        (value posttest )
        (value prebody prebody_function)
        (value postbody postbody_function )
        (value patstart )
        (value patstop )
        (value title )
        (value title2 )
        (value waves )
        (value calset )
        (value driver1 )
        (value driver0 )
        (value driveroff DQS)
        (value beforepat )
        (value duringpat )
        (value waitflagstrue )
        (value waitflagsfalse )
        (value floatpins )
        (value startpat )
        (value stoplabel )
        (value tnum1 40)
        (value forcespec1 )
        (value tnum2 60)
        (value forcespec2 )
)
```

```
(define-instance io_hi_leakage pin_pmu_Template
        (value mpause 0.0005)
        (value lolimval )
        (value hilimval )
        (value levels leakage_levels)
        (value pinlist DQ0 DQ1 DQ2 DQ3)
        (value fload 0 :choice “off”)
        (value irange 15 :choice “20UA”)
        (value measure 1 :choice “Current”)
        (value sequence 1 :choice “seq”)
        (value autotitle 0 :choice “no”)
        (value forceval2 )
        (value forceval1 5.5)
        (value lolimspec IL_lo)
        (value hilimspec IL_hi)
        (value forcespec2 )
        (value tnum2 )
        (value forcespec1 )
        (value tnum1 90)
        (value stoplabel )
        (value startpat )
        (value floatpins )
        (value waitflagsfalse )
        (value waitflagstrue )
        (value duringpat )
        (value beforepat )
        (value driveroff )
        (value driver0 ADDRS DQS)
        (value driver1 RAS CAS WE OE)
        (value calset )
        (value waves )
        (value title2 )
        (value title )
        (value patstop )
        (value patstart )
        (value postbody postbody_function )
        (value prebody prebody_function)
        (value posttest )
        (value pretest )
)
```

Notes

Leakage test use the pin_pmu_Template just as the continuity tests do. These notes will describe the differences but will not repeat the notes under the continuity test. Review the notes on page 3.1.8 if needed.

The **sequence** parameter is set to ganged instead of sequence in the first instance example. This means that the result of all pin PMU measurements will be ORed. The datalog will record the results of this OR operation. A failure of any individual PMU measurement will result in the failure of the ganged result.

Both instances uses the **lolimspec** and **hilimspec** parameters instead of lolimval and hilimval. These parameters will reference the spec database (specadr, specdb) to assign the appropriate values. To find the spec, the instance will locate the level object named by the **levels** parameter. This is leakage_levels, which is defined in the group clause of the workspace database file:

```
(level          leakage_levels
                        (
                        leakage_levs
                        levels_eqns
                        )
)
```

The leakage_levs keyword is the name of a specset. This will be found in the d1Mx4_levels.specadr file. Within that specset the IL_lo and IL_hi specs are defined.

Note that the **pinlist** parameters selects INPUTS for the input_leakage_ganged instance. This is a pingroup consisting of the address. However, the io_hi_leakage template specifies the IO pins independently in the pinlist. Either approach is valid.

In both instances, the **measure** parameter is set to current; this means the **forceval** is a voltage. Notice that the input_leakage_ganged instance uses two forcevals forceval1 is set to 0 volts, and forceval2 is set to 5.5 volts. The template will first execute the test on all input pins at 0 volts and then at 5.5 volts. Since two tests are run two test numbers are needed for the datalog. The **tnum** parameters specify test numbers and, as expected, two are specified: tnum1 set to 40 and tnum2 set to 60.

The first instance sets the IO pins to HiZ using the **driveroff** value. The second instance has a more elaborate setup. The address and IO pins are set to VIL using the **diver0** parameter and the clock pins are set to VIH using the **driver1** parameter. The numerical value of these settings are defined by the levels parameter.

Using Leakage Instances

Using Leakage instances is done just like other instances: you create the instance using Insted, import the instance into your main.t file, and execute it in the test flow using the EXECUTE macro. See page 3.1.5 for an example.

ICC Tests—Background Information

This section will provide background about Icc tests. The ivs_Template is commonly used for Icc tests. The ivs_Template uses the current monitors that are part of the VS hardware to make measurements. It is the easiest of the ivs templates and is recommended for cases where the expected current is at least 10 mA. For tests less than 10 mA, the ivs_monpin_Template or the ivs_pmu_Template could be used (refer to the **J990 Series Templates Reference Manual**).

Test Description

Icc tests are used to detect excess current through device power supply pin(s). In a Icc test, the pattern generator is used to place the input pins in a continuously changing sequence of states. Therefore, expect a pattern parameter will need to be programmed in the instance. A voltage (Vcc) is forced on the power pin(s), and the *averaged* resultant current is measured. This means that the test will need to be repeated several times. The instance includes a parameter, samples, which repeats the test a specified number of times.

For both tests, the measured current value should be less than that specified for the device.

The system provides two resources that can be used to perform static and dynamic ICC tests:

- ❒ The voltage source current monitor
- ❒ The system PMU

Current Monitor for ICC Tests

Power supply current is easily measured with the current monitor.

The only connections required are the hardwired connections between the power pin of the device and the VS force and sense lines on the DIB. As Figure 3.1-6 shows, the current monitor is already set up in the J990 Series mainframe to make the measurement.

Static ICC Test Procedure

To measure static ICC using the current monitor:

1. Program the VS to the appropriate force value.
2. Program a minimum pause time (1ms or less) to let the device power up. (This additional time may be required to charge the bypass capacitors.)
3. Program the drivers to a known state.
4. Perform the test and specify high and low pass limits.

The current monitor for the specified VS is enabled, causing it to measure the current through the force line resistance on the output side of the VS buffer.

The A/D converter reads the analog current value and generates a digital equivalent, which is then compared with the programmed high and low limits in the **test** statement. If the digital value falls outside this range, a failure is recorded. The **test** statement automatically datalogs the results.

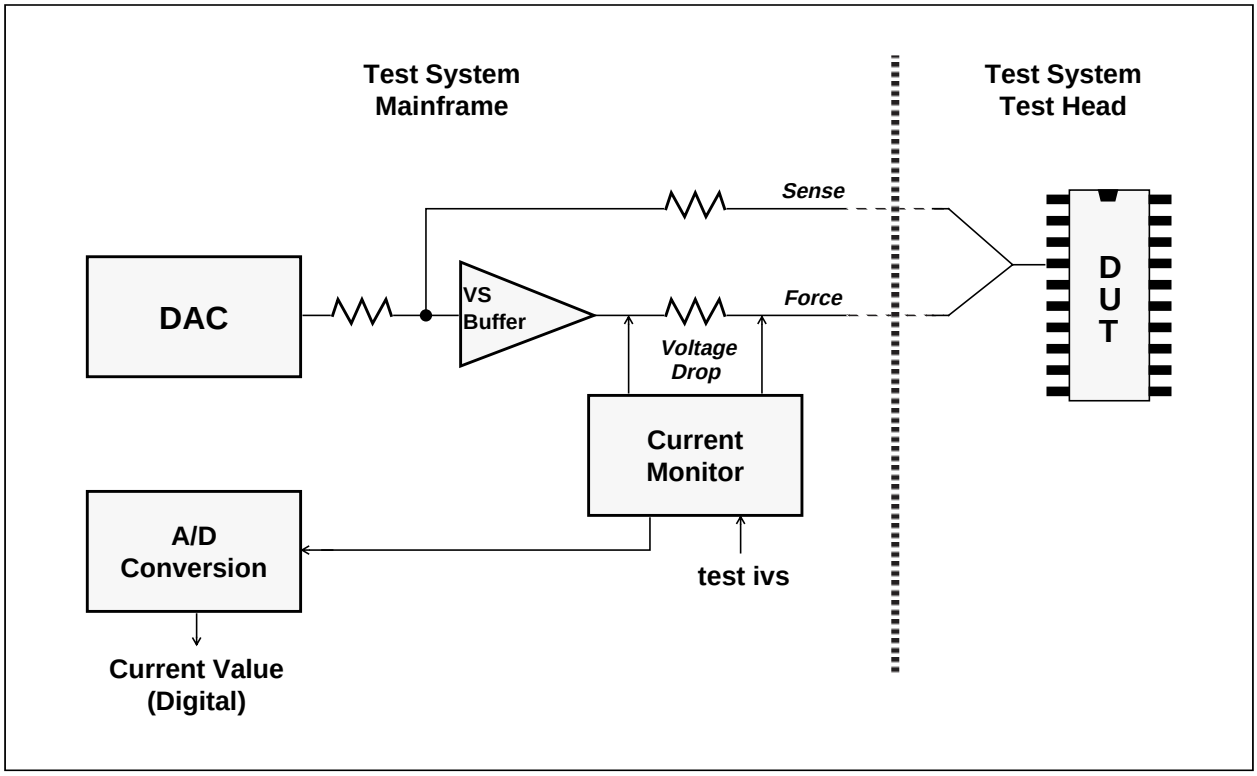


Figure 3.1-6 Voltage Source Current Monitor

Dynamic ICC Test Procedure

For dynamic ICC tests, the pattern generator is used to exercise the device continuously.Then with voltage applied to the power pin(s), the average resultant ICC is measured.

To measure ICC using the current monitor:

1. Select the appropriate pattern from the pattern memory. It may be necessary to create a pattern with an infinite loop to ensure continuous execution of the pattern generator. The pattern should not enable the comparators.
2. Program the VS to the appropriate force value.
3. Start the pattern generator.
4. Program a minimum pause time (1ms or less). (This additional time may be required to charge the bypass capacitors.)
5. Specify high and low pass limits and perform the test. Stop the pattern generator when done.

System PMU for ICC Tests

The system PMU is used in much the same way as the current monitor. The only major differences are in the connections that must be made. (Figure 3.1-7 shows hardware connections prior to ICC testing.)

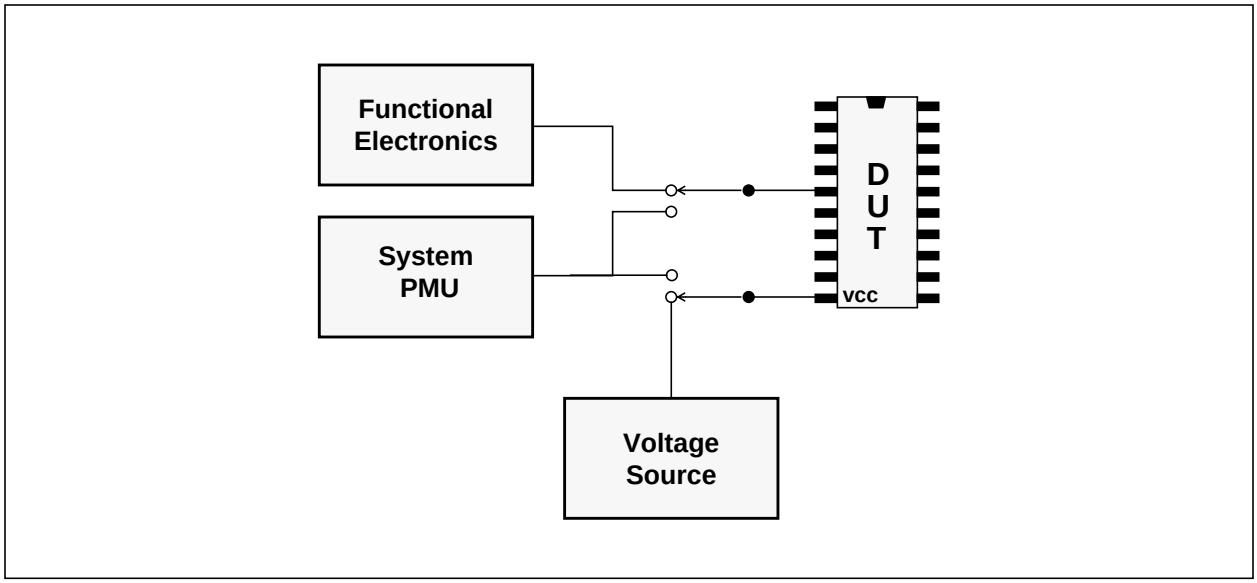


Figure 3.1-7 Hardware Connections Prior to ICC Testing

Static ICC Test Procedure

To measure power supply current with the system PMU resource:

1. Power down the device (remove all loads and input voltages).
2. Turn off the voltage source for the pin to be tested.
3. Connect the system PMU to the first power pin to be tested.
4. Program the PMU to the appropriate force value and specify high and low pass limits.
5. Program the drivers to a known state.
6. Perform the test.
7. Reconnect the power pin to the voltage source.
8. Turn on the voltage source.

Dynamic ICC Test Procedure

Dynamic ICC tests using the system PMU is similar to the same test using the current monitor.

To measure dynamic ICC using the system PMU:

1. Disable the comparators. This will prevent the compare logic from detecting functional errors and possibly stopping the pattern generator prior to the end of the test.
2. Select the appropriate pattern from the pattern memory. It may be necessary to create a pattern with an infinite loop to ensure continuous execution of the pattern generator.
3. Program the PMU to the appropriate force value and specify high and low pass limits.
4. Start the pattern generator.
5. Program a minimum pause time (1ms or less). (This additional time may be required to charge the bypass capacitors.)
6. Perform the test.
7. Stop the pattern generator.

ICC Instances

The following are example ICC instances.

Instance Examples

```
(define-instance icc_ras_refresh ivs_Template
        (value mpause 0)
        (value duringpat refresh_loop_patt)
        (value lolimspec )
        (value hilimspec Iccr)
        (value lolimval 0.001)
        (value hilimval )
        (value forceval 5.5)
        (value ivspin VCC)
        (value irange 0 :choice “1 A”)
        (value samples 10)
        (value waves wave_timing_nominal)
        (value levels icc_levels)
        (value autotitle 0 :choice “no”)
        (value main_alt 0 :choice “main”)
        (value forcespec )
        (value tnum 103)
        (value floatpins )
        (value waitflagsfalse )
        (value waitflagstrue )
        (value beforepat )
        (value driveroff )
        (value driver0 )
        (value driver1 )
        (value calset icc_level_cal)
        (value title )
        (value patstop )
        (value patstart )
        (value postbody postbody_function )
        (value prebody prebody_function)
        (value posttest )
        (value pretest )
)
```

```
(define-instance icc_operating ivs_Template
        (value mpause 0.001)
        (value forceval 5.5)
        (value lolimval 0.001)
        (value hilimval )
        (value lolimspec )
        (value hilimspec Icc)
        (value ivspin VCC)
        (value irange 0 :choice “1 A”)
        (value samples 10)
        (value duringpat read_write_loop_patt)
        (value pretest )
        (value posttest )
        (value prebody prebody_function)
        (value postbody postbody_function )
        (value patstart )
        (value patstop )
        (value title )
        (value calset icc_level_cal)
        (value driver1 )
        (value driver0 )
        (value driveroff )
        (value beforepat )
        (value waitflagstrue )
        (value waitflagsfalse )
        (value floatpins )
        (value tnum 102)
        (value forcespec )
        (value main_alt 0 :choice “main”)
        (value autotitle 0 :choice “no”)
        (value levels icc_levels)
        (value waves wave_timing_nominal)
)
```

Notes

The **mpause** parameter is explained on page 3.1.10.

```
(value duringpat refresh_loop_patt)
```

The **duringpat** parameter is used to specify a pattern that is usually designed to loop endlessly. For this instance, the pattern will be started before the Icc test is done and repeat throughout the execution of the instance. When the instance is finished, the pattern will be stopped.

Here the refresh_loop_patt pattern is executed. The next instance uses a read–write loop pattern. Both of these are patterns are pattern objects as defined in the group clause of the workspace database (wspadr) file. For example,

```
(pattern        refresh_loop_patt
                        (
                        refresh_loop     ;startset
                        dram_patterns    ;mpgset
                        pattern_data     ;dgenset
                        srcset           ;srcset
                        d1Mx4_mut        ;mutset
                        )
)
```

The **hilimspec** and **lolimspec** parameters are described on page 3.1.30. The **hilimval** and **lolimval** parameters are described on page 3.1.9. the **forceval** parameter is described on page 3.1.9. Note, however, that this forceval refer specifies the voltage to be forced by the voltage sources not the pin PMU. Voltage sources have a range of -20.0 to 20.0 volts.

```
(value ivspin VCC)
(value irange 0 :choice “1 A”)
(value samples 10)
```

The **ivspin** parameter is used to specify the pin on which current is to be measured, and, if specified, the voltage is to be forced. The pin named (VCC in this case) is defined in the pinset clause, part of the pin database (pinadr, pindb).

The **irange** parameter is discussed on page 3.1.9.

The **samples** parameter specifies the integer number of current samples to be taken by the current monitor. Here 10 is specified. Note that the default is 80 samples. If this number is not desirable, specify another number here.

The **waves** parameter is described on page 3.1.22. The **levels** parameter is described on page 3.1.10. The **autotitle** parameter is described on page 3.1.5.

```
main_alt 0 :choice “main”)
```

The voltage sources can be programmed to main or alt. The **main_alt** parameter specifies which one is to be used for the forceval. The choices are 0 (main) or 1 (alt). The default is main.

The remaining parameters used in the examples have been explained earlier: **tnum** on page 3.1.30, **calset** on page 3.1.21, **postbody** and **prebody** on page page 3.1.11.

Using Icc Instances

Using Icc instances is done just like other instances: you create the instance using Insted, import the instance into your main.t file, and execute it in the test flow using the EXECUTE macro. See page 3.1.5 for an example.

Chapter 3.2: Customizing Templates

Creating Custom Templates

Introduction

While the previous chapter in this section gave instructions on how to use the standard Teradyne templates and implement instances in a test program, this chapter describes the process of creating and modifying templates to create templates specific to the user— that is, custom templates. Although the standard templates supplied by Teradyne have been designed for the utmost flexibility, some test scenarios may require the use of a template customized to the user's particular need.

The goals of this chapter are to describe how AT900 commands combine to compose tests in a test program, to understand template files and how they interact, and to understand the rules and symbols involved in writing a template.

The tools for writing a template are the Unix VI or Sun's Text Edit. Additional examples beyond those provided in this chapter can be found in `/j99x/templates`.[1]

1. The tester root directory is `/j99x` where j99x is the name of test system used, i.e. j997. It is always lowercase. For example, on a J997 Memory Test System the template directory is located at `/j997/templates`. In a J994 Memory Test System the template directory is located at `/j994/templates`.

Overview of Process

There are several steps to customizing templates. Two files need to be created (or existing approximations edited) to create a custom template. These files are the *.tdl* (template description language) file and the *.t* (template source code) file. The *.h* file (header file) that is included in a the *.t* file is produced automatically from the *.tdl* file through the **tdl_to_h** command. The *.tdl* file is the file used by Insted when it displays and edits instance parameters. The *.t* file is the template source code that is executed when an instance of that template is executed with the EXECUTE macro.

1. **Select the standard Teradyne template closest in structure to the template you are trying to create.** Teradyne strongly recommends that the user copy an existing template to another directory and modify the copy to meet the specific requirement (always renaming the new template to some other name) rather than creating a new template from scratch.

WARNING: NEVER MODIFY THE STANDARD TERADYNE TEMPLATES IN /j99x/TEMPLATES. ONLY MODIFY A COPY!

---

2. Edit the Teradyne template.tdl file to meet the specific requirements for the custom template parameter list.

   Always save the new *template.tdl* file to another (unique) name.

3. Run **tdl_to_h** on the new .tdl file to create the .h file for inclusion into the new template.t file for the custom template.

   ```
   % tdl_to_h CUSTOMNAME_Template.tdl
   ```

   The output file name will be CUSTOMNAME_Template.h.

4. Rename the template.t file to a new filename.t (use same prefix as given to the *.tdl* file in step 2). It is very important to not confuse the standard Teradyne templates with a custom template.
5. Edit the template.t file to perform the requirements of the custom template.
6. Compile the new template.t code with AT900 compiler. For example:

   ```
   %  at900 -g -c basic_func_Template.t
   ```

   produces basic_func_Template.o which is included in the job's workspace to be part of the test program. If you did not add `/j99x/templates` into the path to include Tlib.h into the .t file, then you should use the `-I/j99x/`templates switch on the at900 command line to compile the .t file.

7. Place the new .tdl, .h, .t, and .o files in a directory which can be accessed by the users of the system.
8. Include the new .o file in your workspace file (.wspadr).(Optional) The default .tdl path for the Insted tool is /j99x/templates. So Insted can locate custom templates automatically, enter the following commands:

```
% setenv IG_TEMPLATE_PATH /home/HOSTNAME/custom :/home/HOSTNAME/custom
```

The **setenv** command allows Insted to also look in /home/<hostname>/custom for .tdl files. For convenience, add the setenv command to a .login file.

9. (Optional) Add the following command to the .tdbxtool file:

% use . /j99x/templates /j99x/source/exec_templates /home/HOSTNAME/custom /other

This will allow you to access the source code of Teradyne templates, Teradyne Executive templates, custom templates, and other files from tdbxtool.

10. (Optional) Add the custom template into a library of templates, such as libtemplates.o, which is part of the standard Template collection, by entering:

% ld -r -o libcusttemp.o [CUSTOM.o]$_{1+}$

The line above produces a single .o file that can be added to the test program workspace. This is a simpler method of listing the custom templates in the workspace than manually entering the name of each custom template in the workspace file.

Creating the Custom Template

The first step is selecting the appropriate template to modify from the list of standard Teradyne templates in /j99x/templates. Once that is done, then the programmer must edit the .tdl file and the .t file.

Editing the .tdl File

The .tdl file is simply a list of the parameters that are used in the template. The user of the template who is programming instances will have to fill in the parameters listed (unless marked optional). The main purpose of the .tdl file is to format the Insted editor window so that parameters are listed in a certain order, determine what the required parameters are, define the minimum or maximum value, and determine whether there is a default value if no value is present in the instance of that template.

The syntax for the .tdl file is as follows:

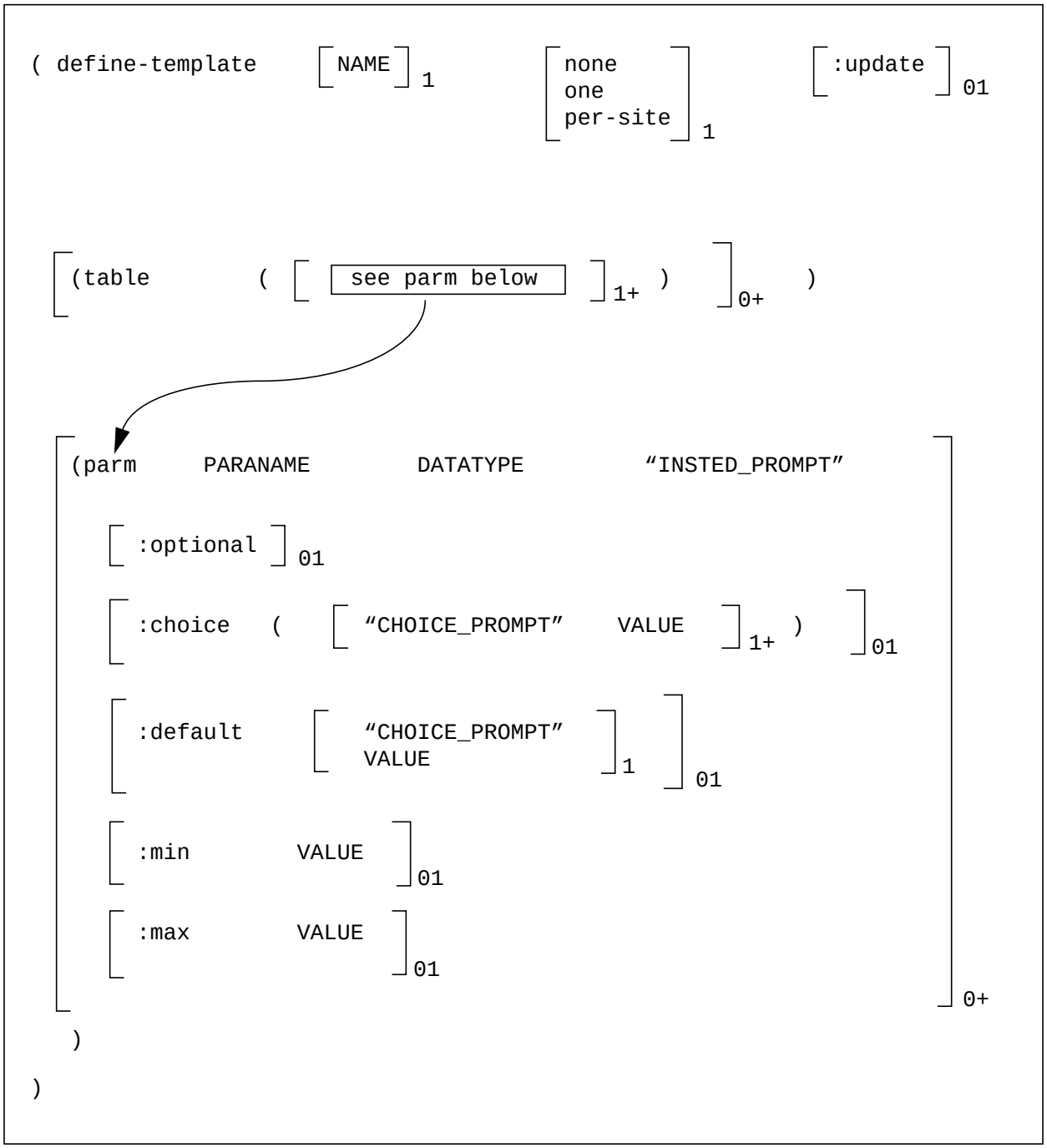


Figure 3.2-1 DEFINE-TEMPLATE Syntax

(DEFINE-TEMPLATE ...) Statement contains the following:

NAME is the name of the template followed by a modifier that defines the type of result that will be received. The options are as follows:

none	The template returns no result.
one	The template returns one result, regardless of the number of devices being tested.
per-site	The template returns a result per site.

:update Calls the Update section of the .t file (see page 3.2.10).

(TABLE ...) Statement Allows for the grouping of several parameter statements in one template, so multiple instances can be executed within one template.

(PARM ...) Statement Can be used with or without the *(table...)* statement. Contains the following:

PARANAME can be any alphanumeric name. Note that the name selected here will be the name of the variable used in the template .t file for that object.

Note: Teradyne strongly recommends that the if a parameter fulfills the same functions as a parameter already used in a Teradyne template, it should use the same name. The purpose is to maintain a consistent interface between Teradyne and custom templates.

DATATYPE is the type of parameter. Acceptable values are:

Cfuncptr	Pointer to a C function returning an integer
Double	Double precision floating point variable
Int	Integer variable
Label	SVM or LVM vector location
Level	Level object constructed in workspace.
Pattern	Pattern object constructed in workspace
Pin	Pin symbol from pinset
Spec	Spec from specset
String	ASCII Text surrounded by quotes " "
Waveform	Waveform object constructed in workspace

Putting a set of parentheses ( ) around the datatype indicates that the parameter contains a list of values for that DATATYPE.

"INSTED_PROMPT" is a string of text surrounded by quotes containing the parameter prompt that will appear in Insted or the .insadr file.

:optional indicates that the entering of a value for this parameter is optional. If *:optional* does not appear, then a value is taken from ***:choice***, ***:default***, ***:min***, or ***:max***, which are discussed below.

:choice ("CHOICE_PROMPT" VALUE) provides a list of valid values for the parameter. The strings in quotes appear in the Insted tool. An example:

```
   (parm   attribute Int   "Attributes"
      :choice   (
         "D0" 1
         "D1" 2
         "D2" 3
         "D3" 4
         "C1" 5
         "C2" 6 )
)
```

If a parenthesis is placed around the VALUE, then one of two syntax options can be used. Identical to the syntax above, the first syntax allows the user to select one or more choices:

```
   (parm   attribute (Int)   "Attributes"
      :choice   (
         "D0" 1
         "D1" 2
         "D2" 3
         "D3" 4
         "C1" 5
         "C2" 6 )
)
```

The second syntax option allows user one selection from a list of choices, each of which are composed of several values:

```
   (parm   attribute (Int)   "Attributes"
      :choice   (
         "D1 D2" (2 3)
         "D1 D2 D3" (2 3 4)
         "C1 C2" (5 6)  )
)
```

:default "CHOICE_PROMPT" is the default value that will be assumed if a choice is not entered. An example:

```
   (parm   stor_mode        Int   "FVM storage mode"
        :choice ("all" 0 "vector" 1 "fails only" 2
                 "vector & fails" 3)
        :default "all"
    )
```

:min <VALUE>and ***:max <VALUE>*** display in the Insted tool the minimum and maximum values for the parameter. An example:

```
(parm   pipeline_length  Int   "Patgen pipeline length?"
   :optional
   :min 230
   :max 249
)
```

Example .tdl File

The following .tdl file, basic_func_Template.tdl, was created as a simpler version of the existing func_Template.tdl on page 3.2.8 , which runs functional tests on multiple sites using the patterns listed in the pattern parameter of the instance. The basic_func_Template is designed for the user who wants to run a functional test that would use only one pattern. The pattern management parameters have been removed. The example below will print the test title, select a waveform, level, and calset, and run the pattern.

```
(define-template basic_func_Template one
   (parm autotitle In "Use Automatic Datalog Title?"
               :choice ("no" 0 "yes" 1)
               :default "no")
   (parm title String "Datalog Title"
               :optional )
   (parm levels  Level    "Levels Name"
               :optional )
   (parm waves Waveform  "Waveform Name"
               :optional )
   (parm calset String  "Calset Name"
               :optional )
   (parm pattern Pattern "Pattern Name" )
   (parm tnum Int "Test number"
               :optional
               :min 1
               :max 65535 )
)
```

```
(define-template func_Template per-site

   (parm   levels               Level     "Levels Name"
           :optional )
   (parm   waves                Waveform  "Waveform Name"
           :optional )
   (parm   calset               String    "Calset Name"
           :optional )
   (parm   patterns             (Pattern) "Pattern Name(s)" )
   (parm   skip                 (Pattern) "Pattern Names(s) To SKIP"
           :optional)
   (parm   trap                 (Pattern) "Pattern To TRAP On"
           :optional)
   (parm   dotests              Int       "do \"test func\" ?"
           :choice ("yes" 1 "no" 0 "fails only" 2)
           :default "yes" )
   (parm   stop_on_first_fail   Int       "Stop on first failing pattern?"
           :choice ("yes" 1 "no" 0 )
           :default "no" )
   (parm   tnum                  (Int)    "Test Number(s)"
           :optional
           :min 1
           :max 65535 )
   (parm   autotitle            Int       "Use Automatic Datalog Title?"
            :choice ("no" 0 "yes" 1)
            :default "no")
   (parm   title                String    "Datalog Title"
           :optional )
   (parm   driver1              (Pin)     "Set Driver = 1 Pinlist"
            :optional )
   (parm   driver0              (Pin)     "Set Driver = 0 Pinlist"
            :optional )
   (parm   driveroff            (Pin)     "Set Driver Off Pinlist"
            :optional )
   (parm   pretest              Cfuncptr  "Pretest Function"
           :optional )
   (parm   posttest             Cfuncptr  "Posttest Function"
           :optional )
   (parm   prebody              Cfuncptr  "Prebody Function"
           :optional )
   (parm   postbody             Cfuncptr  "Postbody Function"
           :optional )
   (parm   patstart             Cfuncptr  "Patgen Start Function"
           :optional )
   (parm   patstop              Cfuncptr  "Patgen Stop Function"
           :optional )
)
```

Editing the Template .t File

In order to edit the .t file to create a custom template, several concepts and sets of rules need to be understand. The concepts that will be covered in this section are:

1. General concepts that apply to all template customization.
2. General description of the structure of the template source code.
3. Description of template macros defined by Teradyne.
4. Description of template Tlib variables and their usage.
5. Declaring template C++ variables
6. How to write statements for error checking, etc.

General Recommendations

It is far more efficient to modify an existing Teradyne template than to create a new template from scratch. By modifying an existing template from `/j99x/templates`, the programmer can make use of example code already present and have code that follows all of the recommendations listed in this chapter already in front of him.

Template Source Code Structure

The source code is partitioned into five major sections, all of which **must** be present in the source code .t file.

Includes — This section will typically have the following three include statements. teradyne.h will always be present in any template. The last include statement will be the .h file that was created from running tdl_to_h on the .tdl file earlier in the chapter.

```
#include <teradyne.h>
#include "Tlib.h"
#include "CUSTOMNAME_Template.h"
```

#include teradyne.h must always be present in any template to include Teradyne specific settings needed for compile and load operations.

#include "/Tlib.h" is required for using Teradyne macros, which are used to write .t file code. To add the correct path in the #include statement use the *-I/j99x/templates* switch when the .t file is compiled with the AT900 compiler. This simply places */j99x/templates/* in front of *Tlib.h*.

#include "CUSTOMNAME_Template.h" must always be present. It is the name of the .h file created from running tdl_to_h

Template Identifiers — This section has just two statements. Both contain the name of the template you are writing. For the example template, this section looks like:

```
/*  Internal template variables, arrays, functions */
static char templatename[] = "CUSTOMNAME_Template";
/*  State Template I.D.   */
TEMPLATE_ID(CUSTOMNAME_Template);
```

The rest of the template code is contained in three sections: prebody, body, and

postbody. This format organizes the template in to three parts: the first part containing one-time setups, the second part containing steps that perform tests, and the third part containing clean up code from the setup.

Update Section (Optional)— This section contains code that can be called either when a job is loaded or through Insted's Update messages. Although there are no restrictions as to what code may be placed in this section, it is most often used for caching information to make instances run faster.

Prebody Section — This section is used for one time setups for a template. This section performs actions like selecting the levels, waveforms, calsets, print the test title, etc. Note that the prebody section is **not** the same as the prebody function which can be an argument to a template. The prebody function is an interpose function which gives the user a chance to add in his own code as the first step of the body section in a template.

Body Section — This section performs the statements involved in performing the actual measurement or test. This section must contain the code required to make each measurement valid in the template. That means that if code is required to precondition a device before any leakage test, then that code should be present in this section.

Postbody Section — This section is used for one time cleanups from a template. The code present in this section would restore connections to hardware from a certain test when done, check for alarms after the test, clear any on_patgen_start, on_patgen_stop, on_pretest, or on_posttest functions. Note that the postbody section is **not** the same as the postbody function which can be an argument to a template. The postbody function is an interpose function which gives the user a chance to add in his own code as the last step of the body section in a template.

When an instance is executed, the Prebody, Body, and Postbody section are run sequentially. The Update section is usually run when a value(s) has been changed, because either a job was loaded (i.e., the hardware changes from not having value settings to having value settings) or a value was edited during a debug session.

Template Code Overview

There are several different symbols and languages that work together to form a template.

1. AT900 statements
2. C/C++ code
3. Teradyne Tlib.h macros already defined for the user
4. Template library calls/variables for the user.

Teradyne macros are various combinations of code created to make template writing easier and less error prone. These macros are for the most part one-for-one substitutions done by the preprocessor during compiling or calls to Teradyne functions performed during compiling. Macros often begin with TEMPLATE_, TLIB_ or TT_ and are always upper case.

The library of variables and functions are predefined by Teradyne. When a program is compiled, variables insert values and functions insert sets of code. Teradyne variables often begin with Tlib_ .

As separate C++ and test system manuals provide comprehensive documentation on the AT900 and C++ statements available for the user, this chapter focuses on creating templates using AT900 and the template macros and library calls. Teradyne strongly recommends that a parameter used for the same purpose as a parameter in the Teradyne standard templates should use the same name so as to present a consistent template interface.

FUNC_TEMPLATE.T

The .t file for the Func Template is shown on the following two pages.

```
#include <teradyne.h>
#include "func_Template.h"
#include "Tlib.h"
static char templatename[] = "func_Template";
TEMPLATE_ID(func_Template);
TEMPLATE_PRE_BODY_BEGIN(func_Template)
      TLIB_TEMPLATENAME;
      TLIB_TIMER(TLIB_STARTIME,TLIB_PREBOD);
      TLIB_LEVEL(levels);
      TLIB_WAVEFORM(waves);
      TLIB_CALDATA(calset);
      TLIB_AUTOTITLE(autotitle);
      TLIB_TITLE(title);
      TLIB_PRETEST(pretest);
      TLIB_POSTTEST(posttest);
      TLIB_PATSTART(patstart);
      TLIB_PATSTOP(patstop);
      TLIB_CHECK_DISABLE;
      Tlib_initialize_siteset ();
      TLIB_TIMER(TLIB_STOPTIME,TLIB_PREBOD);
TEMPLATE_PRE_BODY_END(func_Template)
TEMPLATE_BODY_BEGIN(func_Template)
      Pattern V;
      Pattern S;
      Pin P;
      Int T;
      Instance *Ip;
      int flag;
      int failflag;
      TLIB_TIMER(TLIB_STARTIME,TLIB_BOD);
      Ip = readsys(instance);
      if (HASVALUE(prebody))
          (*prebody)();
      flag = 0;
      for (V = patterns, T = tnum; HASVALUE(V); NEXT(V),NEXT(T)) {
            if (dotests) {
                  if (HASVALUE(T)) {
                        if (T > 0)
                    writesys(testnumber,T);
                  }
                  else if (flag) {
                        int x = readsys(testnumber);
                        writesys(testnumber,x + 1);
                  }
            }
            if (HASVALUE(skip)) {
                  flag = 0;
                  for (S = skip; HASVALUE(S); NEXT(S)) {
                        if (S == V) {
                              flag = 1;
                              break;
                        }
                  }
                  if (flag)
                        continue;
            }
            if (HASVALUE(driver1)) {
                        for (P = driver1; HASVALUE(P); NEXT(P)) {
                  writesys(set_driver_mode,P,1);
                  writesys(set_driver_state,P,1);
            }
      }
```

```
63.         if (HASVALUE(driver0)) {
64.                     for (P = driver0; HASVALUE(P); NEXT(P)) {
65.                     writesys(set_driver_mode,P,1);
66.                     writesys(set_driver_state,P,0);
67.                   }
68.           }
69.         if (HASVALUE(driveroff)) {
70.               for (P = driveroff; HASVALUE(P); NEXT(P)) {
71.                     writesys(set_driver_mode,P,0);
72.               }
73.         }
74.                     writesys(patgen_start, V);
75.                     failflag = 0;
76.                     writesys(patgen_wait);
77.                     if (readsys(cr)) {
78.                           failflag = 1;
79.                           JOB_SERIAL {
80.                                 if (readsys(cr)) {
81.                                 Tlib_remove_site ();
82.                                 if (dotests == 0)
83.                                  writesys(instance_result,*Ip,TT_CM_FAIL);
84.                                 }
85.                           }
86.                     }
87.                     switch (dotests) {
88.                           case 0://NEVER
89.                                 break;
90.                           case 2://IF PATTERN FAILS
91.                                 if (failflag == 0)
92.                                 break;
93.                           case 1://ALWAYS
94.                           default:
95.                                 writesys(test_func);
96.                                 break;
97.                     }
98.                     if (HASVALUE(trap)) {
99.                           for (S = trap; HASVALUE(S); NEXT(S)) {
100.                                if (S == V)
101.                                Tlib_trap (S);
102.                          }
103.                    }
104.              if (stop_on_first_fail && failflag) {
105.                    if (Tlib_active_sites_count () < 1)
106.                          break;
107.              }
108.        }
109.        Tlib_alarmcheck ();
110.        if (HASVALUE(postbody))
111.              (*postbody)();
112.        TLIB_TIMER(TLIB_STOPTIME,TLIB_BOD);
113.  TEMPLATE_BODY_END(func_Template)
114.  TEMPLATE_POST_BODY_BEGIN(func_Template)
115.        TLIB_TIMER(TLIB_STARTIME,TLIB_POSTBOD);
116.          Tlib_pretest_clear();
117.          Tlib_posttest_clear();
118.          Tlib_patstart_clear();
119.          Tlib_patstop_clear();
120.        TLIB_TIMER(TLIB_STOPTIME,TLIB_POSTBOD);
121.  TEMPLATE_POST_BODY_END(func_Template)
```

Lines 1 - 3 Initialization Statements

Initialization Statements (Lines 1 - 3)

Line	Code	Type	Description
1.	`#include <teradyne.h>`	C++	Include the teradyne.h file during the preprocessing stage.
2.	`#include "func_Template.h"`		Include the func_Template.h file during the preprocessing stage.
3.	`#include <Tlib.h>`		Include the Tlib.h file during processing.

Lines 4 - 5 Template Identifiers

Template Identifiers (Lines 4 - 5)

Line	Code	Type	Description
4.	`static char template-name[] = "func_Template";`	C++	Declare that the template name is func_Template.
5.	`TEMPLATE_ID(func_Template);`	Macro	Declare the name of the template for the test system Executive or operating system.

Line 6 - 21 Prebody Section

Prebody Section (Lines 6 - 21)

Line	Code	Type	Description
6.	`TEMPLATE_PRE_BODY_BEGIN (func_Template)`	Macro	Mark the beginning of the Prebody section.
Lines 7-19 hold one-time setup information that is loaded during the preprocessing stage.			
7.	`TLIB_TEMPLATENAME`	Macro	Initialize the template name for the Executive and template software.
8.	`TLIB_TIMER (TLIB_STARTIME, TLIB_PRE_BOD);`		Starts a timer that measures the execution time of the prebody and datalogs the results.
9.	*`TLIB_LEVEL(levels)`*		Select a new levels object defined in the workspace file.
10.	*`TLIB_WAVEFORM(waves)`*		Select a new waveform object defined in the workspace file.
11.	*`TLIB_CALDATA(calset)`*		Select a new calset.
12.	*`TLIB_AUTOTITLE(autoti-tle)`*		Print the template and instance names in the test program datalog if autotitle parameter is yes.
13.	*`TLIB_TITLE(title)`*		Print the current instance's *title* parameter.

Prebody Section (Lines 6 - 21)

Line	Code	Type	Description
14.	`TLIB_PRETEST(pretest)`	Macro	Set up the calling of a function (C++ function returning an integer) after the initial test command setup has occurred and before the test command occurs. The *on* condition setup remains in effect until cleared in the *TEMPLATE_POST_BODY* section via a call to *Tlib_pretest_clear()* which resets the function pointer to the function in effect before the instance was executed. Typically, the set up of any *on_pretest* conditions should be performed only in the *TEMPLATE_PRE_BODY* section.
15.	*`TLIB_POSTEST(posttest)`*		Set up the calling of a function (C++ function returning and integer) after actual test command has occurred but before the value is datalogged and marked for pass/fail. The "on" condition setup will remain in effect until cleared in the *TEMPLATE_POST_BODY* section via a call to *Tlib_postest_clear()* which resets the function pointer to the function in effect before the instance was executed. Typically, the set up of any *on_pretest* conditions should be performed only in the *TEMPLATE_PRE_BODY* section.
16.	*`TLIB_PATSTART(patstart)`*		Set up the call of a function (C++ function returning an integer) before the patgen is started (most often used for device preconditioning before the vectors run). The "on" condition setup will remain in effect until cleared in the TEMPLATE_POST_BODY section via a call to *Tlib_patstart_clear()* which resets the function pointer to the function in effect before the instance was executed. Typically, setting up any *on_patgen_start* conditions should be performed only in the *TEMPLATE_PRE_BODY* section.
17.	*`TLIB_PATSTOP(patstop)`*		Set up the call of a function (C++ function returning an integer) after the patgen is stopped (used for post-conditioning). The "on" condition setup will remain in effect until cleared in the *TEMPLATE_POST_BODY* section via a call to *Tlib_patstop_clear()* which resets the function pointer to the function in effect before the instance was executed. Typically, the setting up any *on_patgen_start* conditions should be performed only in the *TEMPLATE_PRE_BODY* section.
18.	*`TLIB_CHECK_DISABLE`*		Disable the comparator for pins designated in the *disable_comp_pins* parameter in the instance for the Background Template of the test program. It is normally executed once at the end of the TEMPLATE_PRE_BODY section.
19.	*`Tlib_initialize_siteset`*	Function	Declares to the template library a list of the initial sites.
20.	*`TLIB_TIMER`* `(TLIB_STOPTIME, TLIB_PRE_BOD);`	Macro	Stops a timer that measures the execution time of the prebody and datalogs the results.
21.	`TEMPLATE_PRE_BODY_END()`		Mark the end of the Prebody section.

Line 22 - 109 **Body Section**

The Body section is where the actual test instructions reside. It consists of preliminary setup information and a main *for* function.

Preliminary Setup Information Lines (22 - 34)

Line	Code	Type	Description
22.	`TEMPLATE_BODY_BEGIN (func_template)`	Macro	Mark the beginning of the Body section.
23.	`Pattern V`	AT900	Declare that V is a pattern object.
24.	`Pattern S`		Declare that S is a pattern object.
25.	`Pin P`		Declare that P is a pin symbol, which can be a list of device pins or a pin group.
26.	`Int T`		Declare that T is of Teradyne integer data type.
27.	`Instance *IP`		Declare that Ip is a pointer to an instance.
28.	`int flag;`	C++	Declare flag as an integer type.
29.	`int failflag`		Declare failflag as an integer type.
30.	`TLIB_TIMER(TLIB_STARTIME, TLIB_BOD);`	Macro	Starts a timer that measures the execution time of the body and datalogs the results.
31.	`Ip = readsys(instance);`	AT900	Load the instance in the pointer location *Ip and assign it to Ip.
32.	`if (HASVALUE(prebody))`	Macro	If a function exists in the prebody parameter of the current instance, go to line 34; if not, go to line 35.
33.	`(*prebody)();`	AT900	Execute the function specified by the prebody parameter in the current instance.
34.	`flag=0`	C++	Declare that the variable flag is equal to zero or false.

Main FOR Function (Line 35)

```
for (V = patterns, T = tnum; HASVALUE(V); NEXT(V),NEXT(T)) {
```

The next line is the beginning of a *for* function. Line 35 analyzes the current instance loaded in line 31. It uses the values in the instance's *patterns* and *tnum* fields to determine the outcome of conditional code, which activates various statements. The *for* function holds many loops (described later on). Line 36 is divided into three sections:

1. *V = patterns, T = tnum;* defines the initial states for the fields in the executed instance.

2. *HASVALUE(V)* declares that if *V* has a value, then go to the next line. If there are no patterns (or all of the patterns values have been already analyzed), then go to the end of the *for* function (line 108).

3. *NEXT(V), NEXT(T)* are increment instructions to select the next *patterns* and *tnum* value from the instance loaded in memory after the main *for* loop has been executed.

Below is a description of each line in the main *for* function of func_Template.t. ***Please note that the missing lines are beginning or end braces in the code.***

Main FOR Function (Lines 35 - 108)

Line	Code	Type	Description
35.	`for (V = patterns, T = tnum; HASVALUE(V); NEXT(V),NEXT(T))`	C++/ Macro	See page 3.2.16
36.	`if (dotests)`		If the *dotests* parameter value is 1 (true) in the instance, go to the next line; if not, go to line 46.
37.	`if (HASVALUE(T))`		If the *tnum* parameter value exists in the instance, then go to the next line; if not, go to line 41
38.	`if (T > 0)`	C++	If T is greater than 0, execute the next statement; if not, go to line 46.
39.	`writesys(testnumber,T);`	AT900	Write the *tnum* value to the Executive for datalogging.
41.	`else if (flag)`	C++	If the *flag* parameter value is 1 in the instance, go to next step; if it is 0, go to line 46.
42.	`int x = readsys(testnumber);`	C++/ Macro	Read the current test number, assign it to x, and declare it as an integer.
43.	`writesys(testnumber, x + 1);`	AT900	Add 1 to the test number.
46.	`if (HASVALUE(skip))`	C++/ Macro	If the current instance has a skip parameter, go to the next step; if not, go to line 57.
47.	`flag = 0;`	C++	Reset the flag to 0 or false.
48.	`for (S = skip; HASVALUE(S); NEXT(S))`	C++/ Macro	Define S as equal to *skip* (`S = skip`). If S has a value then go to the next step; if not, go to line 54 (`HASVALUE(S);`). Select the next *skip* value (`NEXT(S)`).
49.	`if (S == V)`	C++	If S has the same value as V, go to the next line; if not, go to line 48 and select the next *skip* value (`NEXT(S)`)
50.	`flag = 1;`		Set the flag to 1 (true).
51.	`break;`		Get out of the *for* loop started in line 48 and go to line 54.
54.	`if (flag)`		If flag is true (0) go to the next line; if not, go to line 57.
55.	`continue;`		Go to line 34 and get the next *skip* parameter value in the current instance.

Main FOR Function (Lines 35 - 108)

57.	`if (HASVALUE(driver1))`	C++/ Macro	If the current instance has a *driver1* parameter value, go to the next step; if not, go to line 63.
58.	`for (P = driver1; HASVALUE(P); NEXT(P))`		Define P as equal to *driver1* (`P = driver1`). If P has a value, go to the next step; if not, go to line 63 (`HASVALUE(P);`). Select the next *driver1* value (`NEXT(P)`).
59.	`writesys (set_driver_mode,P,1);`	AT900	Set the driver mode to on for the pins specified by P or *driver1.*
60.	`writesys (set_driver_state,P,1);`		Set the driver initial logical level to 1 for the start of a burst. Then go to line 58 and select the next *driver1* value (`NEXT(P)`).
63.	`if (HASVALUE(driver0))`	C++/ Macro	If the current instance has a driver0 parameter, go to the next step; if not, go to line 69.
64.	`for (P = driver0; HASVALUE(P); NEXT(P))`		Define P as equal to *driver0* (`P = driver0`). If P has a value, go to the next step; if not, go to line 69 (`HASVALUE(P);`). Select the next *driver0* value (`NEXT(P)`).
65.	`writesys (set_driver_mode,P,1);`	AT900	Set the driver mode to on for the pins specified by P or *driver0.*
66.	`writesys (set_driver_state,P,0);`		Set the driver initial logical level to 0 for the start of a burst, go to line 64, and select the next *driver0* value (`NEXT(P)`).
69.	`if (HASVALUE(driveroff))`	C++/ Marco	If the current instance has a *driveroff* parameter, go to the next step; if not, go to line 74.
70.	`for (P = driveroff; HASVALUE(P); NEXT(P))`		Define P as equal to *driveroff* (`P = driver0`). If P has a value, go to the next step; if not, go to line 74 (`HASVALUE(P);`). Select the next *driveroff* value (`NEXT(P)`).
71.	`writesys (set_driver_mode,P,0);`	AT900	Set the driver initial logical level to 0 for the pin specified by P or *driveroff*, go to line 70, and select the next *driveroff* value (`NEXT(P)`).
74.	`writesys (patgen_start, V);`		Start the pattern generator using the current pattern value.
75.	`failflag = 0;`	C++	Set failflag to 0 (false).
76.	`writesys(patgen_wait);`	AT900	Wait for patgen to stop.
77.	`if (readsys(cr))`	C++/ Macro	Return an *int* type indicating failure(s) for the last pattern burst. If it is true a failure(s) occurred, go to the next line 78; if it is false, go to line 87.
78.	`failflag = 1;`	C++	Set failflag to 1 (true).
79.	`JOB_SERIAL`	AT900	Run a specified block of code on devices one at a time instead of running the block of code on the devices in parallel.
80.	`if (readsys(cr))`	C++/ Macro	Return an *int* type indicating a failure for the last pattern burst. If it is true that a failure occurred, go to line 81; if false, go to line 79 and the next device.
81.	`Tlib_remove_site ();`	Function	Remove current site from the active site list.
82.	`if (dotests == 0)`	C++	If the *dotests* parameter is the same as the value 0, go to line 83; if not, go to line 79 and the next device.
83.	`writesys(instance_result, *Ip, TT_CM_FAIL);`	AT900/ Macro	Write the current site result as fail in the current instance data structure.

Main FOR Function (Lines 35 - 108)

87.	`switch (dotests)`	C++	Get the *dotest* parameter value in the current instance.
88.	`case 0:`		If the *dotest* value is 0 (i.e., don't do test func), go to the next line; if not, go to line 90.
89.	`break`		Get out of *switch* statement and go to line 98.
90.	`case 2:`		If the *dotest* value is 2 (i.e., do test func if fails only), go to the next line; if not, go to line 93.
91.	`if (failflag == 0)`		If failflag is the same value as 0 (false), go to line 92; if not, go to line to 93.
92.	`break;`		Get out of *switch* statement and go to line 98.
93.	`case 1:`		If the *dotest* value is 1 (i.e., do test func), go to the next line; if not, go to line 94.
94.	`default:`		The *dotest* value is neither 1, 2, nor 0, go to the next line.
95.	`writesys(test_func);`	AT900	Start pattern burst in the current instance, get the pass/fail information, and datalog the results.
96.	`break;`	C++	Get out of the *switch* statement and go to line 98.
98.	`if (HASVALUE(trap))`	C++/ Macro	If the current instance has a trap parameter, go to the next step; if not, go to line 104.
99.	`for (S = trap; HASVALUE(S); NEXT(S))`		Define S as equal to *trap* (`S = trap`). If S has a value, go to the next step; if not, go to line 104 (`HASVALUE(S);`). Select the next *trap* value (`NEXT(S)`).
100.	`if (S == V)`	C++	If S has the same value as V, go to line 101; if not, go to line 99 and select the next trap value (`NEXT(S)`).
101.	`Tlib_trap (S);`	Function	Trap on the value in S (*trap*) in the current pattern, go to line 99, and select the next trap value (`NEXT(S)`).
104.	`if (stop_on_first_fail && failflag)`	C++	If *stop_on_first_fail* and *failflag* are true, go to line 105; if they are not, go to line 35 and select the next pattern in the *pattern* parameter.
105.	`if (Tlib_active_sites_count () < 1)`	C++/ Function	If there are no more active sites (see line 81), go to the next line; if not, go to line 35 and select the next pattern in the *pattern* parameter.
106.	`break;`	C++	Get out of the main for loop and go to line 109.

Lines 109 - 113 Remaining Lines in the Body Section

Body Section Continued (Lines 109 - 113)

Line	Code	Type	Description
109.	`Tlib_alarmcheck ();`	Function	Check for VS Kelvin or overload alarms and return message if there are such problems.
110.	`if (HASVALUE(postbody))`	C++/ Macro	If a function exists in the Postbody parameter of the current instance, go to line 111; if not, go to line 112.
111.	`(*postbody) ();`		Execute the function specified by the Postbody parameter in the current instance.
112.	`TLIB_TIMER(TLIB_STOPTIME ,TLIB_BOD);`	Macro	Stops a timer that measures the execution time of the body and datalogs the results.
113.	`TEMPLATE_BODY_END(func_t emplate)`		Mark the end of the Body section.

Lines 114 - 121 Postbody Section

Post Body Section (Lines 114 - 121)

Line	Code	Type	Description
114.	`TEMPLATE POST_BODY_BEGIN (func_Template)`	Macro	Mark beginning of Postbody section.
115.	`TLIB_TIMER(TLIB_STARTIME ,TLIB_POSTBOD);`		Starts a timer that measures the execution time of the postbody and datalogs the results.
116.	`Tlib_pretest_clear();`	Function	Clear pretest setup conditions.
117.	`Tlib_posttest_clear();`		Clear postest setup conditions.
118.	`Tlib_patstart_clear();`		Clear patstart setup conditions.
119.	`Tlib_patstop_clear();`		Clear patstart setup conditions.
120.	`TLIB_TIMER(TLIB_STOPTIME ,TLIB_POSTBOD);`	Macro	Stops a timer that measures the execution time of the postbody and datalogs the results.
121.	`TEMPLATE POST_BODY_END (func_Template)`		Mark the end of the Postbody section.

C++ Variable Declarations

Often variables need to be defined in template code in addition to the variables defined through the processing of the .tdl file. Typically these are variables used to increment through loops or store values. As an example:

```
if ((!(HASVALUE(beforepat))) && HASVALUE(floatpins)) {
        Pin P; // increment variable of type Pin required.

        for (P = floatpins; HASVALUE(P); NEXT(P)) {
             Tlib_floatpins(P);
        }
        writesys(pause_for_relay);
   }
```

In this example, beforepat is a pattern object. Floatpins is a list of Pin symbols. Both are instance parameters set to values using Insted.

In this example, a variable of type Pin was required in order to loop through pin symbols in a *for* loop. Variables can be defined just about anywhere in the template code. The following high-level types are defined for you and match the list of datatypes available when defining parameters in the .tdl file.

Template Macros

The following is a list of the macros used to write templates. These macros make template writing easier and less prone to error. For additional examples beyond those mentioned in this chapter, review the existing standard templates in `/j99x/templates`.

```
TEMPLATE_PRE_BODY_BEGIN(   )
TEMPLATE_PRE_BODY_END(   )
TEMPLATE_BODY_BEGIN(   )
TEMPLATE_BODY_END(   )
TEMPLATE_POST_BODY_BEGIN(   )
TEMPLATE_POST_BODY_END(   )
```

These symbols mark the beginnings and ends of each of the three sections of the template code. These macros **must** be present. The symbol placed in the parenthesis is the name of the template. For example,

```
TEMPLATE_PRE_BODY_BEGIN(basic_func_Template)
TEMPLATE_PRE_BODY_END(basic_func_Template)
```

```
TEMPLATE_RETURN;
```

This macro is the way to return control to the test program from the template without executing any more code in that particular section of the template. Template code cannot employ the standard C return construct `return;`. For example:

```
TEMPLATE_BODY_BEGIN(basic_func_Template)

        Instance *Ip;
         Ip = readsys(instance);

        if (HASVALUE(pattern))
               //standard template parameter names used.
            writesys(patgen_start, pattern);
        else {
            TLIB_EPR ("No pattern to run specified!!");
            TLIB_FORCETEMPLATEFAIL;
            TEMPLATE_ABORT;
            TEMPLATE_RETURN;
        }
        if (HASVALUE(tnum))
              //standard template parameter names used.
            writesys(testnumber,tnum);
        writesys(test_func);
        TEMPLATE_RETURN;

TEMPLATE_BODY_END(basic_func_Template);
```

`TEMPLATE_ABORT;`

This macro is used to abort template code in the middle of execution. This kind of abort usually accompanies some kind of error check — for instance, if an error exists, the template will flag the error and stop execution. This macro returns control to the test program. TEMPLATE_ABORT sets a flag that indicates to the software that the entire template should be aborted, not just the present section of the template at the next TEMPLATE_RETURN statement. When an immediate abort is desired after a designated error, TEMPLATE_RETURN should always follow TEMPLATE_ABORT. This is shown in the TEMPLATE_RETURN example above.

`TEMPLATE_ID();`

This macro is used to identify the template for the test system executive. The template name is placed here as an argument. This macro appears before the start of the template code.

```
TEMPLATE_ID(basic_func_Template);
```

`TLIB_CALDATA();`

This macro is used to select a calset. This macro has built-in error checking, making it a better choice than the `writesys(select_calset,...)` statement. Typically, selecting of new calset objects should be performed only in the TEMPLATE_PRE_BODY section. For example, the following statement would be used to select the calset,

```
TLIB_CALDATA("my_calset");  // string is the parameter
                            // passed.
```

`TLIB_FORCETEMPLATEFAIL;`

This macro is used to indicate that execution of the template produced a failure of some kind. This can be used to flag a failure of execution of the template due to errors encountered in the data in the instance or to flag failures in patgen bursts, or elsewhere even though no tests were actually performed (see the example shown in the TEMPLATE_RETURN section above.)

`TLIB_LEVEL();`

This macro is used to select a new levels object. It accepts an argument of type level, an object defined in the workspace file. This macro has extensive built-in error checking, making it a superior choice to the `writesys(select_level,...)` statement. Typically, selecting of new levels objects should be performed only in the TEMPLATE_PRE_BODY section. (This is not true for the corner_list_Template.) For example, the following statement would be used to select the level,

```
TLIB_ LEVEL(levels); // standard Teradyne parameter names used.
```

`TLIB_WAVEFORM();`

This macro is used to select a waveform object. It accepts an argument of type waveform, an object defined in the workspace file. This macro has built-in error checking that makes it superior to the `writesys(select_waveform,...)` statement. Typically, selecting of new waveform objects should be performed only in the TEMPLATE_PRE_BODY section. (This is not true for the corner_list_Template.) For example, the following statement would be used to select the waveform,

```
TLIB_WAVEFORM(waveforms);  // standard Teradyne parameter
                           // names used.
```

`TLIB_AUTOTITLE();`

This macro is used to cause automatic titles to print the template name and the instance name used in the test program datalog. Typically, printing of titling information should be performed only in the TEMPLATE_PRE_BODY section. For example,

```
TLIB_AUTOTITLE(autotitle); //standard Teradyne parameter
                           // names used.
```

`TLIB_TITLE();`

This macro is used to print out any user defined string to label the instance. This can be in addition to or instead of using the autotitle feature described above.

Typically, printing of titling information should be performed only in the TEMPLATE_PRE_BODY section. For example,

```
TLIB_TITLE(title); // standard Teradyne parameter names
                   // used.
```

`TLIB_PRETEST();`

This macro is used to call a function (C function returning an integer) after initial test command setup has occurred but before the actual test command action occurs. The *on* condition setup will remain in effect until cleared in the TEMPLATE_POST_BODY section through a call to `Tlib_pretest_clear()`, which resets the function pointer to the function in effect before the instance was executed. Typically, setting up `on_pretest` conditions should be performed only in the TEMPLATE_PRE_BODY section. For example,

```
TLIB_PRETEST(pretest); // standard Teradyne parameter names
                       // used.
```

`TLIB_POSTTEST();`

This macro calls a function (C function returning an integer) after an actual test command has occurred, but before the value is datalogged and marked for pass/fail. The *on* condition setup will remain in effect until cleared in the TEMPLATE_POST_BODY section through a call to `Tlib_posttest_clear()`, which resets the function pointer to the function in effect before the instance was executed. Typically, setting up `on_posttest` conditions should be performed only in the TEMPLATE_PRE_BODY section. For example,

```
TLIB_POSTTEST(posttest); // standard Teradyne parameter
                         // names used.
```

`TLIB_PATSTART();`

This macro is used to call a function (C function returning an integer) before the patgen is started (most often used for device preconditioning before the vectors run). The *on* condition setup will remain in effect until cleared in the TEMPLATE_POST_BODY section through a call to `Tlib_patstart_clear()`, which resets the function pointer to the function in effect before the instance was executed. Typically, setting up `on_patgen_start` conditions should be performed only in the TEMPLATE_PRE_BODY section. For example,

```
TLIB_PATSTART(patstart);  //standard Teradyne parameter
                          // names used.
```

`TLIB_PATSTOP();`

This macro is used to call a function (C function returning an integer) after the patgen is stopped (useful for postconditioning). The *on* condition setup will remain in effect until cleared in the TEMPLATE_POST_BODY section through a call to `Tlib_patstop_clear()`, which resets the function pointer to the function in effect before the instance was executed. Typically, setting `on_patgen_start` conditions should be performed only in the TEMPLATE_PRE_BODY section. For example,

```
TLIB_PATSTOP(patstop);  //standard Teradyne parameter
                        // names used.
```

`TLIB_CHECK_DISABLE;`

This macro is used to disable the comparator on pins designated in the disable_comp_pins parameter in the instance for the background_Template of the test program. It is normally executed once at the end of the TEMPLATE_PRE_BODY section.

`HASVALUE();`

This macro is used to check to see whether the parameter has a value in the instance or not. Instances which have optional parameters must use this macro to check for to see if a value is present before acting on it. This macro is used throughout all sections of a template for this purpose. For example,

```
if (HASVALUE(tnum))
      writesys(testnumber,tnum);
```

In this example, the testnumber is only changed if the tnum parameter in the instance has a value.

`NEXT();`

This macro is used to increment through a list of objects for a particular parameter.

In the following example, the variable P is defined to be of type Pin and is used to increment through a list of pins. The driver1 parameter to the template is defined with parentheses around it in the .tdl file, indicating that the parameter can take a list of pins symbols as its argument. In this example, the outer loop first checks to see if there is a list of pins. If there is a list, the Pin variable is set to the first pin symbol in the driver1 list. The drivers are set to the high state. Then the Pin variable P is incremented to point to the next entry in the driver1 list. If there is a pin symbol, the same operation is performed on that pin. This *for* loop is typical; most will have a similar structure for incrementing through lists in template code.

```
Pin P;

if (HASVALUE(driver1)) {
      for (P = driver1; HASVALUE(P); NEXT(P)) {
           writesys(set_driver_mode,P,1);
           writesys(set_driver_state,P,1);
      }
}
```

`TLIB_EPR (“”);`

This macro is used for outputting an error from the template code to the datalog.For example,

```
if (HASVALUE(duringpat)) {
      if (readsys(patgen_busy))
            writesys(patgen_stop);
      else
        TLIB_EPR (“\”duringpat\” stopped before testing was complete.\n”);
}
```

`TLIB_DEB;`

The macro TLIB_DEB is used to output a formatted string. For example,

```
if (fval > 40.0 || fval < -40.0) {
        sprintf(Tlib_ebuf,
        “force value %7.2lf V out of range, aborting.\n”,fval);
        TLIB_DEB;
        TLIB_FORCETEMPLATEFAIL;
        TEMPLATE_ABORT;
        TEMPLATE_RETURN;
}
```

Tlib Variables/Functions

The following is a list of global variables used in template code to interface with template software and the test system executive. All template macro and variable definitions are in Tlib.h unless otherwise specified.

Tlib_minforce/Tlib_maxforce

These global variables are of type double and are set to the maximum and minimum values by the force range of the PMU. These variables are set in the `Tlib_set_pmu_force_bounds()` function and used for reference.

Tlib_tnumarray[2][TEMPLATE_MAXCHANS]

This array is used to store test numbers for pin PMU and system PMU tests. The first half of the array (Tlib_tnumarray[0][...]) is used for the test numbers of the first of 2 setups for parametric tests (say low voltage associated with the tnum1 parameter of the pin_pmu_Template). The second half of the array (Tlib_tnumarray[1][...]) is associated with the second of 2 setups for parametric tests (say high force voltage). This array is used to process test numbers for the PMU and PPMU Templates.

Tlib_debug

This variable is an integer type and is used to flag whether the template code is being debugged or not. The user can program some statements to output additional status messages to the screen if this variable is set. Then, if Tlib_debug is set to 1 in tdbxtool using the command *`assign Tlib_debug = 1`* to put the template code in the special mode, more debug data will be outputted. For example,

```
Spec S = refspec;

if (Tlib_debug)
      PR "\n Spec = \"%s\"",
               readsys(spec_symbol,S));
```

Tlib_specs_modified

This variable is of integer type and is used to track whether the template code has changed the value of any spec in the spec database or not. This statement must always accompany any `writesys(spec_value, ...)` statement in the template code. Whenever values are modified, they should first be saved in a storage place so that they can be restored at the end of the template. When modifying the spec value, set `Tlib_specs_modified = 1`. When restoring the original value, reset `Tlib_specs_modified = 0`. For example,

```
writesys(spec_value,SPEC,waves, newvalue);
        Tlib_specs_modified = 1;

        writesys(spec_value,SPEC,waves, restorevalue);
                Tlib_specs_modified = 0;
```

Tlib_set_pmu_force_bounds(measure, vrange, irange);

This function, which resides in Tlib.o, is used to set the maximum and minimum force values, Tlib_maxforce and Tlib_minforce, for given voltage and current ranges of the PMU. The function is of type *void* and does not return a value. Acceptable values for the function parameters are:

`measure = 0` (measure voltage) or `1` (measure current),

```
vrange = TT_PMU_RANGE_40V,
         TT_PMU_RANGE_20V, or
         TT_PMU_RANGE_2V,
```

and

```
irange = TT_PMU_RANGE_200MA,
         TT_PMU_RANGE_20MA,
         TT_PMU_RANGE_200UA,
         TT_PMU_RANGE_20UA, or
         TT_PMU_RANGE_2UA
```

Tlib_pmu_rngstring(measure, irange, vrange, forcevalue);

This function is used to help print out information about errors in the force value of the PMU or pin PMU. The function takes four parameters:

`measure = 0` (measure voltage), `1` (measure current)

```
vrange =    TT_PMU_RANGE_40V,
            TT_PMU_RANGE_20V,
            TT_PPMU_RANGE_20V, or
            TT_PMU_RANGE_2V

irange =    TT_PMU_RANGE_200MA,
            TT_PMU_RANGE_20MA,
            TT_PMU_RANGE_2MA,
            TT_PPMU_RANGE_2MA,
            TT_PMU_RANGE_200UA,
            TT_PPMU_RANGE_200UA,
            TT_PMU_RANGE_20UA,
            TT_PPMU_RANGE_20UA,
            TT_PMU_RANGE_2UA,
            TT_PPMU_RANGE_2UA, or
            TT_PPMU_RANGE_200NA
```

`forcevalue = <the value forced>`
This is a double value.

Tlib_set_ppmu_force_bounds(measure, irange, vrange);

This function, which resides in `Tlib.o`, is used to set the maximum and minimum force values, `Tlib_maxforce` and `Tlib_minforce`, for given voltage and current ranges of the pin PMU. The function is of type *void* and does not return a value.

Acceptable values for the function parameters are:

`measure = 0` (measure voltage) or `1` (measure current),

```
          vrange =   TT_PPMU_RANGE_20V

and       irange =   TT_PPMU_RANGE_2MA,
                     TT_PPMU_RANGE_200UA,
                     TT_PPMU_RANGE_20UA,
                     TT_PPMU_RANGE_2UA, or
                     TT_PPMU_RANGE_200NA
```

Tlib_floatpins(Pinlist)

This function is used to open the connection from the test station to the DUT pin (DUTcon), leaving the pin to float. For example,

```
if ((!(HASVALUE(beforepat))) && HASVALUE(floatpins)) {
        Pin P;

        for (P = floatpins; HASVALUE(P); NEXT(P)) {
                Tlib_floatpins(P);
         }
         writesys(pause_for_relay);
 }
```

It cannot be used for VS connections. Each call to this function saves the connect state on a stack.

Tlib_unfloatpins()

This function is used to close the connection from the test station to the DUT pin, so that the pin no longer floats. For example,

```
Pin P;

for (P = floatpins; HASVALUE(P); NEXT(P)) {
       Tlib_unfloatpins();
}
```

In this example, P is initialized to the first symbol in the floatpins list. The *for* loop continues while there are entries left in the floatpins list. At the end of each iteration through the loop, the Pin pointer is incremented to the next pin symbol in the list. This function must be called once for each call to TLIB_floatpins. It restores values from the pin connection stack.

Tlib_alarmcheck();

This function checks for VS kelvin or overload alarms and reports them with a message. The function is of type *void*, returning no value. This function is useful anytime the user wants to scan for VS alarms.

Tlib_save_levels(Pinlist);

This function saves all functional levels (vih, vil, voh, vol, vch, vcl, ioh, iol, and vt) and stores them in a structure for later data retrieval.

Tlib_zero_levels(Pinlist);

This function sets all functional levels (vih, vil, voh, vol, vch, vcl, ioh, iol, and vt) to 0.

Tlib_restore_levels(Pinlist);

This function restores all functional levels (vih, vil, voh, vol, vch, vcl, ioh, iol, and vt) that were stored during the `Tlib_save_levels` call. The user will get an error if he tries to restore levels that were not previously stored.

Tlib_save_all_vs();

This function saves all VS data (main, alt, ilimit, etc.) and stores them in a structure for later data retrieval.

Tlib_zero_all_vs();

This function sets all VS levels to 0.

Tlib_restore_all_vs();

This function restores all VS levels that were stored during the `Tlib_save_all_vs` call. The user will get an error if he tries to restore levels that were not previously stored.

Tlib_trap(Pattern);

This function can be used to trap on a certain function name just after the program runs the pattern generator burst so that the user can examine the burst more closely.

Tlib_pmu_check_irange(TT_RANGE irange);

This function returns the current range used by the PMU. It returns a default range if the range passed to it is illegal. This function is used in PMU templates to verify that the user's current range is correct.

Tlib_pmu_check_vrange(TT_RANGE vrange);

This function returns the voltage range used by the PMU. It returns a default range if the range passed to it is illegal. This function is used in PMU templates to verify that the user's voltage range is correct.

Tlib_pin_pmu_check_irange(TT_RANGE irange);

This function returns the current range used by the pin PMU. It returns a default range if the range passed to it is illegal. This function is used in pin PMU templates to verify that the user's current range is correct.

Tlib_setstop(Label stoplabel);

This function inserts a stop opcode in the vectors at the label stoplabel passed to the function. It is used to run a patgen burst up to the point the device is preconditioned to the correct state for the intended test. It is used in the standard Teradyne templates in the pin PMU and system PMU templates to designate a stopping point in the `startpat` vector burst at which measurements can be made.

Tlib_clearstop():

This function clears the stop opcode from the vector setup by the last `Tlib_setstop()` function call.

Tlib_get_time();

This function is used to read back the current time. The function returns the current time as an integer number of seconds. It can be used to measure elapsed time as in the following example.

```
int starttime, stoptime, elapsedtime;

          starttime = Tlib_get_time();

                              :
          stoptime = Tlib_get_time();
          elapsedtime = stoptime - starttime;
```

Tlib_start_timer();

This function declared in `Tlib_time.h` is used to start the utility timer which produces a higher resolution time measurement than `Tlib_get_time()`.

Tlib_stop_timer();

This function is used to return the elapsed time on the utility timer since the `Tlib_start_timer()` call.

Important Coding Tips

There are several important items to consider in coding a custom template. This section provides some general observations.

For Loops

When using *for* loops to increment variables of various types use the template macros `HASVALUE()` and `NEXT()`. For example,

```
Pin P;

for (P = floatpins; HASVALUE(P); NEXT(P)) {
Tlib_unfloatpins();
}
```

In this example, `P` is initialized to the first symbol in the floatpins list. The *for* loop continues while there are entries left in the floatpins list (`HASVALUE()`). At the end of each iteration through the loop, the Pin pointer is incremented to the next pin symbol in the list (`NEXT()`)

Background_Template

Even though the user may want to use custom templates, many job setups can be accomplished with an instance of the background_Template. Therefore, Teradyne strongly recommends that users use the background_Template in their test programs if it is possible to do so. Also, IO pins that are input only will have their comparator disabled through the background_Template.

Optional Parameters and Defaults

Many parameters are optional in the templates. The user is not required to enter a value for these parameters. However, often the template code must act on the object for which the parameter entry was intended in the .tdl file. In this case, the template software must have a way of determining default values for optional parameters. There are three specific ways to set up defaults for template parameters.

1. **Parameter value inherited from environment.** This method is the simplest method. The programmer only performs an action on a value if a value is present. Otherwise, he does nothing with any objects of that type. This is the defaulting method used for most parameters. It means that the waveform previously selected is not changed. For example,

   ```
   if (HASVALUE(waves))
       writesys(select_waveform, waves);
   ```

2. **Parameter has hidden default in code.** This method will directly set a value if the instance parameter does not have a value. For example,

   ```
   if (HASVALUE(tnum))
       writesys(testnumber, tnum);
   else
       writesys(testnumber, 1);
   ```

This is another way to set up a default, but it is **not** recommended. It requires the user to look through the template code in order to see what would happen if he did not enter a value.

3. **Use :default in .tdl file.** This is the recommended way of establishing a specific default value for a parameter. The default is documented in the Insted tool and the default can be changed like any other database entry rather than by searching through deeply embedded code. For example,

```
(parm samples    Int     "Samples"
       :default 80
       )
```

If the user does not enter a value for the samples parameter in this example, the value will be 80.

Enums to Variables

Many AT900 commands take only *enumerated* data types, like those defined at the front of the readsys/writesys statement description section of the **Syntax Reference Manual**. However, the user may want to use a variable in the statement since template code is general purpose in its intent. Variables provide flexibility and can save a lot of coding time. To convert a statement that only uses *enums* to one that uses a variable use the same technique shown in the following example:

Normally the `writesys(pmu_gate,...)` statement looks like the following:

```
writesys(pmu_gate,TT_CM_PMU0,TT_CM_ON);
```

which gates PMU 0 on. This statement could also be written to use a variable, in this case an integer variable selected in the instance database:

```
writesys(pmu_gate,TT_CM_VAL((int)(whichpmu)),TT_CM_ON);
```

This statement, using TT_CM_VAL(), allows the *whichpmu* parameter to be a variable whose value is selected in the instance database. Since using TT_CM_VAL means accepting a variable rather than an *enum*, the error checking should be done by the user in the .tdl file by using :min, :max, and :choice. The correct integer values for enums in writesys statements can be found in `/j99x/include/tt_cm.h`.

Preparing the Custom Template Executable Code

Once the new .tdl and .t files have been edited, it is easy to prepare the code for execution.

1. Run tdl_to_h on the .tdl file to create the .h include file. This .h file will then be included in the .t file at compilation of the .t file. For example,

   ```
   % tdl_to_h basic_func_Template.tdl
   ```

   produces basic_func_Template.h which is included in basic_func_Template.t

2. Compile the .t file using the AT900 compiler. For example,

   ```
   %  at900 -g -c basic_func_Template.t
   ```

   produces basic_func_Template.o which is included in the job's workspace to be part of the test program. If you did not add `/j99x/templates` into the path to include Tlib.h into the .t file, then you should use the `-I/j99x/`templates switch on the at900 command line to compile the .t file.

3. (Optional) Add the custom template into a library of templates, such as libtemplates.o, which is part of the standard Template collection, by entering:

   ```
   %  ld -r -o  libcusttemp.o [CUSTOM.o]1+
   ```

   The line above produces a single .o file that can be added to the test program workspace. This is a simpler method of listing the custom templates in the workspace than manually entering the name of each custom template in the workspace file.

Note: Finally, always give a custom template a unique name different than any other standard or custom template name. Start all custom template filenames with '*ct_*'.

Chapter 4.1: Memory Fail Processing (MFP) Overview

Introduction

Memory fail processing software provides support for programming the fail processing hardware (Catch RAM and related hardware) for the purposes of bitmapping, redundancy analysis, and other failure processing functions. The MFP software provides complete low-level syntax for all fail processing hardware in addition to high-level support for bitmapping of most memory devices. The software consists of the following parts:

- ❒ MFP templates
- ❒ MFP database
- ❒ ADR to DB translator
- ❒ DB to ADR reverse translator
- ❒ a set of high-level interface functions which can be called from the job
- ❒ a set of low-level hardware drivers
- ❒ code which generates hardware setups from (high-level) template inputs
- ❒ enhancements to the Bitmap Tool that provide a graphical user interface for MFP software

Note: J990 Series programmers who are not planning to use the Catch RAM or create Bitmaps should still use MFP software to program scramble RAM contents, multiplexer settings, ADS bit assignment, and topological inversion.

Using the MFP Software

Essentially, there are two approaches to using the memory fail processing software. One approach is the *graphical-user-interface method*. New features and buttons for Bitmap tool support MFP operation. Template instances, required with either approach, can be created using Insted. An MFP database is automatically generated from the templates. These databases can be edited with the ADR translator and reverse translator. The ***Bitmap Tool Manual*** describes the use of the graphical user interface.

The second approach is the *job-driven interface method*. In this approach, function calls are included in the main job flow to implement MFP operations. As with the graphical-user-interface method, MFP template instances are required. These MFP templates are discussed in *Chapter 4.2 Memory Fail Processing Templates.*

The job-driven interface method can be further subdivided into two approaches. The first is the high-level approach. The high-level approach uses high-level function calls within the job program to set up the Catch RAM, capture data, and scan the Catch RAM to create Bitmaps. *Chapter 4.3 Memory Fail Processing High-level Programming* will describe this approach.

The second job-driven approach is to use low-level drivers along with hand-tuned MFP databases. This approach can be used for applications that go beyond the capability of the high-level function calls. *Chapter 4.4 Memory Fail Processing Low-level Programming* describes this approach.

MFP Templates

The primary MFP template is the *MFP Setup Template*. An instance of this template ties instances of all the other templates together into one coherent setup. Each distinct fail capture setup is represented by an instance of this template. Many different scans may be done from a single MFP Setup instance, but it will support only one arrangement of Catch RAM slices (and Scramble RAM contents, multiplexer settings, ADS bit assignment, etc.) for fail capture.

In the MFP Setup instance, you specify the names of all of the other instances which should be part of a particular capture setup. Insted views all of these parameters (the other template instances) as optional, since it is possible to specify only the name of an ADR file (which will bypass the high-level setup generation code entirely). However, assuming you do not wish to override the setup generation with your own ADR input file, most of the instance parameters become mandatory. The following list describes the parameters for the MFP setup instance:

- ❒ MFP MUT Description Template Instance (mandatory)
- ❒ MFP Address Scramble Template Instance (mandatory)
- ❒ MFP Bitmap Usage Template Instance (optional)
- ❒ MFP Scan Time Address Scramble Instance (optional)
- ❒ MFP MUT Topology Template Instance (optional; may be a list of different scan topologies)
- ❒ MFP Data Generator Instance (mandatory)

If no topology instance is specified, a default one will be generated which will lay out IO's side by side.

There are two different forms of the address scramble and topology templates. One form is based on per address-bit equations and is used for devices for which the number of rows, columns, and IOs are all even powers of two. The alternate form (addr_scram2 and topology2) is based on address mapping functions included in the user's job, and it is used for devices for which the number of rows, columns, or IOs are not an even power of two. The user must use the same form (equation or function based) for both templates — they cannot be mixed.

For a full description of each of the templates see Chapter 4.2.

Parallel Capture

In the bitmap usage template, you specify the number of devices that you wish to capture in parallel. The MFP software will attempt to generate a setup that allows the capture of that many parts simultaneously. However, if based on the device size, testing speed, IO channel selection (J994 only), Catch RAM option, and desire to do masking or accumulative bitmapping, there is insufficient Catch RAM available, the MFP code will reduce the number of parallel sites and display a warning message.

The setups for parallel capture are described in terms of pseudo sites rather than physical sites. Pseudo sites are numbered by MFP software starting with pseudo site number 0 and incrementing up by one until the number of sites specified in the bitmap usage template have been mapped to Catch RAM hardware (or less if Catch RAM memory is not available) Physical sites are numbered the same as in the chanadr file (specified by *site N*). The concept of pseudo sites allows the user to generate a setup with any number of devices in parallel, and use it with various different active site lists without needing to modify or regenerate the setup. The MFP drivers look at the active site mask when called and assign the first N active sites to the N pseudo sites.

Mask/Accumulate Support

There is support for setting up the Catch RAM in a *split* mode for two types of applications: masked/unmasked bitmaps, and current test/accumulated tests bitmaps. For masked/unmasked bitmaps, the *mask* (usually hard fails) is stored in one half of the Catch RAM while all fails (hard plus the *soft* fails brought out by characterization) are stored in the other half of the Catch RAM. During the scanning phase of a bitmap plot, the hard fails are subtracted out (masked) and the plot shows only the soft fails. In the current tests/accumulated tests bitmaps, one half of the Catch RAM gets cleared before each test and stores *current test* fails. The other half gets cleared only when directed by the user and is used to store the accumulated fails of all tests.

These applications are selected in the *Bitmap Usage* template. Only one of these options may be selected at a time. Enabling either option will result in half of the Catch RAM being dedicated for mask or accumulation, reducing by half the number of devices which can be captured in parallel or the size of the bitmaps which can be constructed.

High-Level Programming

For job-driven bitmapping (see *Chapter 4.3 MFP High-Level Programming* for a flow diagram illustrating the job-driven bitmapping process), the high-level driver functions are most appropriate. In conjunction with template instances and the MFP databases automatically derived from the instances, these functions take care of most of the details involved in the creation of useful bitmaps. The function calls assert hardware-load driver sets (hldsets) to accomplish many of their functions. These hldsets, part of the MFP databases (and thus subject to editing using ADR translators), can be programmed independently with either the high-level or low-level programming approach.

Low-Level Programming

When programming with the low-level MFP interface, you are provided with function interfaces for capture *setup* and scan *setup*; however, the actual scan and capture operations are left up to the programmer. There are two versions of both the low-level capture setup and scan setup. One version takes an MFP setup instance name as an argument and the other version takes an mfpadr filename directly. If the low level is called with an MFP setup instance, then the input ADR file specified in that instance is extracted. This input ADR file is the only portion of the instance that will be used; all other fields are ignored.

ADR Files

When the template input or high level function calls are not flexible enough to meet the user's needs, a custom ADR file can be used.

Generally, when you need to create a custom ADR file, start with an MFP Setup template and request that an ADR file be generated based on values in those instances. The generated ADR file can then be used as a starting point and further modified as needed. To get the starter ADR file written out, specify an ADR output file in the MFP Setup instance. Note that this setup generation happens during job run, at the first point where an MFP driver function is called which references that MFP Setup instance, so you need an active job to make this happen.

There are two ways that a hand-tuned (or created from scratch) ADR file may be used: with either high-level functions, or with low-level driver calls.

The simplest case is to use the high-level calls just as they would be used for unmodified, template-generated setups. In this approach, you start with template instances and specify an ADR output file. The generated ADR file is modified as necessary, and then it is specified in the MFP Setup instance as an ADR input file. When a value is given for the ADR input file parameter, the MFP high-level drivers will bypass the normal setup-generation phase and use the contents of that ADR file. In this case, it is important that the edits made to the ADR file do not include modification of hldset names. The high-level drivers determine which hldsets to send to hardware based on operation type and hldset name. In general, the high-level drivers will assert several hldsets for a single action.

The second, and more difficult approach, is to use the custom ADR file in conjunction with the low-level driver calls. With this approach, you specify in each driver call the name of the hldset and a bitmask of one or more form-types (hardware subsystems) which should be sent from that hldset to hardware. In this way you can choose to create large hldsets and assert only a portion of them in a given driver call, or you can create small hldsets and assert them in their entirety, perhaps making multiple driver calls to assert multiple hldsets to accomplish a single operation such as setting up for capture.

Chapter 4.4 Memory Fail Processing Low-Level Programming describes in detail the ADR syntax and use of the translators.

Hldsets

Hldsets (hardware-load driver sets) provide grouping within the MFP ADR file of related hardware areas. For instance, in the setups generated from template input, one hldset contains scramble RAM contents, multiplexer settings, and ADS assignments, since all of these things are interrelated. When you wish to create an ADR file from scratch, you can group things any way you want within the constraints of the transfer-types available in the `tt_mfp_job.h` file. That is, because it is not possible to assert scramble RAM contents without programming multiplexers, etc., there is a transfer type named TT_MFP_HLD_TYPE_SCRAM which groups together such items. Likewise, those same forms should be grouped together in a single hldset since they need to be asserted together.

Setup Generation and Interactive Modifications

Setup generation occurs at *first run* time. That is, the first time you use a particular setup instance, a lot of computation is done, including the building of software caches to speed up subsequent calls. This process takes awhile the first time you run the program, but all subsequent executions are very fast.

The MFP software is also highly interactive. You can change the setups freely without doing a job reload. In general when you make an interactive change, the MFP software will detect the change and do some recomputation and cache rebuilding at the next job run. This will result in a delay the first time the modified job is run, but not on subsequent executions.

You may modify instance values at any time with Insted. The MFP driver functions will detect that instance values have changed and re-run the setup generation code the next time that instance is used. Note, if the next time the instance is used is during an operation by Bitmap tool, there will be a delay in the plotting process.

If you are doing job-driven bitmapping based on a custom ADR file, you may also edit that ADR file at any time. The MFP drivers will detect that the .mfpadr file is newer than its corresponding .mfpdb file and will automatically retranslate, reload database values, and so on the next time the ADR file is referenced. Any translation errors will appear in the job IO window.

MFP Workspace Requirements

MFP objects must be added to the job's workspace file along with other changes as described below.

tt_mfp.svmadr changes

1. Copy the file `tt_mfp.svmadr` from `/j90x/etc` to your job directory and add write permission to it.
2. Edit the file to add the job's pinset name and to force any required pin states in the pre-coded patterns in the file. Pin states only need to be edited if it is important to keep devices in a *disabled* state during Catch RAM clear and scan functions. Do not alter the patterns other than adding pin data if required.
3. Add the tt_mfp.svmadr file to the job makefile.
4. Translate the tt_mfp.svmadr file to the equivalent `tt_mfp.svmdb` file.

tt_mfp.startadr changes

1. Copy the `tt_mfp.startadr` file from `/j99x/etc` to the job's directory. Because no edits to this file should be required, you may alternatively create a link to the tt_mfp.startadr file by typing the following from the job's directory:

   ```
   ln -s /j99x/etc/tt_mfp.startadr
   ```

2. Add the tt_mfp.startadr file to the job's makefile.
3. Translate the `tt_mfp.startadr` file into the `tt_mfp.startdb` file.

workspace.wspadr changes

1. Add the following include line to your workspace:

   ```
   #include "/j99x/etc/tt_mfp_wsp.h"
   ```

2. Add the tt_mfp_wsp_setup to the job group in the job's `workspace.wspadr` file:

   ```
   (group
         (default  :comment nil)

         (job    XXXXXXXX
                 (
                 tt_mfp_wsp_setup
                 )
         )
   )
   ```

3. You may optionally choose to include the tt_mfp_XXX_patterns into an SVM table in the workspace.

Chapter 4.2: Memory Fail Processing (MFP) Templates

Introduction

Memory Fail Processing software provides support for programming the error processing hardware for the purposes of bitmapping. The MFP software provides a complete low-level syntax and high-level support. This chapter describes the high-level templates.

MFP Templates

The primary MFP template is the *MFP Setup Template*. An instance of this template ties instances of all the other MFP templates together into one coherent setup. Each distinct capture setup is represented by an instance of this template. Many different scans may be done from a single MFP Setup instance, but it will support only one arrangement of Catch RAM slices (and Scramble RAM contents, multiplexer settings, ADS bit assignment, etc.) for error capture.

In the MFP Setup instance, you specify the names of all of the otherMFP instances which should be part of a particular capture setup. Insted views all of these parameters (the other template instances) as optional, since it is possible to specify only the name of an ADR file (which will bypass the high-level setup generation code entirely). However, assuming you do not wish to override the setup generation with your own ADR input file, most of the instance parameters become mandatory. The following list describes the parameters for the MFP setup instance:

- ❒ MFP MUT Description Template Instance (mandatory)
- ❒ MFP Address Scramble Template Instance (mandatory)
- ❒ MFP Bitmap Usage Template Instance (optional)
- ❒ MFP Scan Time Scramble Instance (optional)
- ❒ MFP MUT Topology Template Instance (optional; may be list of different scan topologies)
- ❒ MFP Data Generator Template Instance (mandatory)

If no topology instance is specified, a default one will be generated which will lay out IO's side by side.

MFP Address Scramble Template

MFP Address Scramble Template (Insted name: *mfp_addr_scram_Template*) contains address scramble information for the memory-under-test. This template is based on per address-bit equations and is appropriate for devices where the number of rows, columns, or IOs are powers of two.

The MFP Address Scramble Template Pin Table

MFP Address Scramble Template contains an Insted "table" that lists an equal number of

Address pin names
Cycle positions
Scramble expressions

Each horizontal "line" in the table represents scramble information for one cycle of each address pin listed in column 1 of the table.

Parameters

MFP Address Scramble Template has the following parameters, listed **in order of appearance in Insted**.

Table 4.2-1. MFP Address Scramble Template Parameters

Insted Prompt	Parameter	Data Type	Entry Required	Description
Which Patgen?	patgen	Integer	No (defaulted)	Specifies which patgen (MPG/ scrambler) will be used. Choices are 0 (Main), 1 (Alt), and 2 (Both). Default = 0 (Main).
Address Pin Names	addr_pins	Pinlist	Yes*	*Pin table data — column "1" fields* Lists pin names for which scramble information is provided. *For each **addr_pins** entry, there must be corresponding entries in **suffix** and **scram_expression**.
Cycle Position	suffix	List of integers	Yes*	*Pin table data — column "2" fields* Specifies cycle position — X (d0..d2) or Y (d3..d5) half of pin for each pin in **addr_pins**. *For each **addr_pins** entry, there must be corresponding entries in **suffix** and **scram_expression**. Choices are 0 (X), 1 (Y).

Table 4.2-1. MFP Address Scramble Template Parameters (Continued)

Insted Prompt	Parameter	Data Type	Entry Required	Description
Scramble Expression	scram_expression	List of strings	Yes*	*Pin table data — column "3" fields* Contains quoted strings giving the scramble expression for each pin in **addr_pins**. *For each **addr_pins** entry, there must be corresponding entries in **suffix** and **scram_expression**. Each addr_pins entry must have a px or py term (or an expression of them) showing the patgen x and y bits to be mapped to that pin. If no scrambling is desired, use px0 for A0, px1 for A1, etc.

Example The following example instance shows programming for MFP Address Scramble Template.

d12X_10Y_x8_scram instance (mfp_addr_scram_Template)

```
(define-instance  d12X_10Y_x8_scram  mfp_addr_scram_Template
   (value patgen 0 :choice "Main")
   (value addr_pins
      A0  A1  A2  A3  A4
      A5  A6  A7  A8  A9  A10 A11
      A0  A1  A2  A3  A4
      A5  A6  A7  A8  A9
   )
   (value suffix
      0   0   0   0   0
      0   0   0   0   0   0   0
      1   1   1   1   1
      1   1   1   1   1
   )
   (value scram_expression
      "px0^py1" "px1" "px2" "px3" "px4"
      "px5"     "px6" "px7" "px8" "px9" "px10" "px11"
      "py0"     "py1" "py2" "py3" "py4"
      "py5"     "py6" "py7" "py8" "py9"
   )
)
```

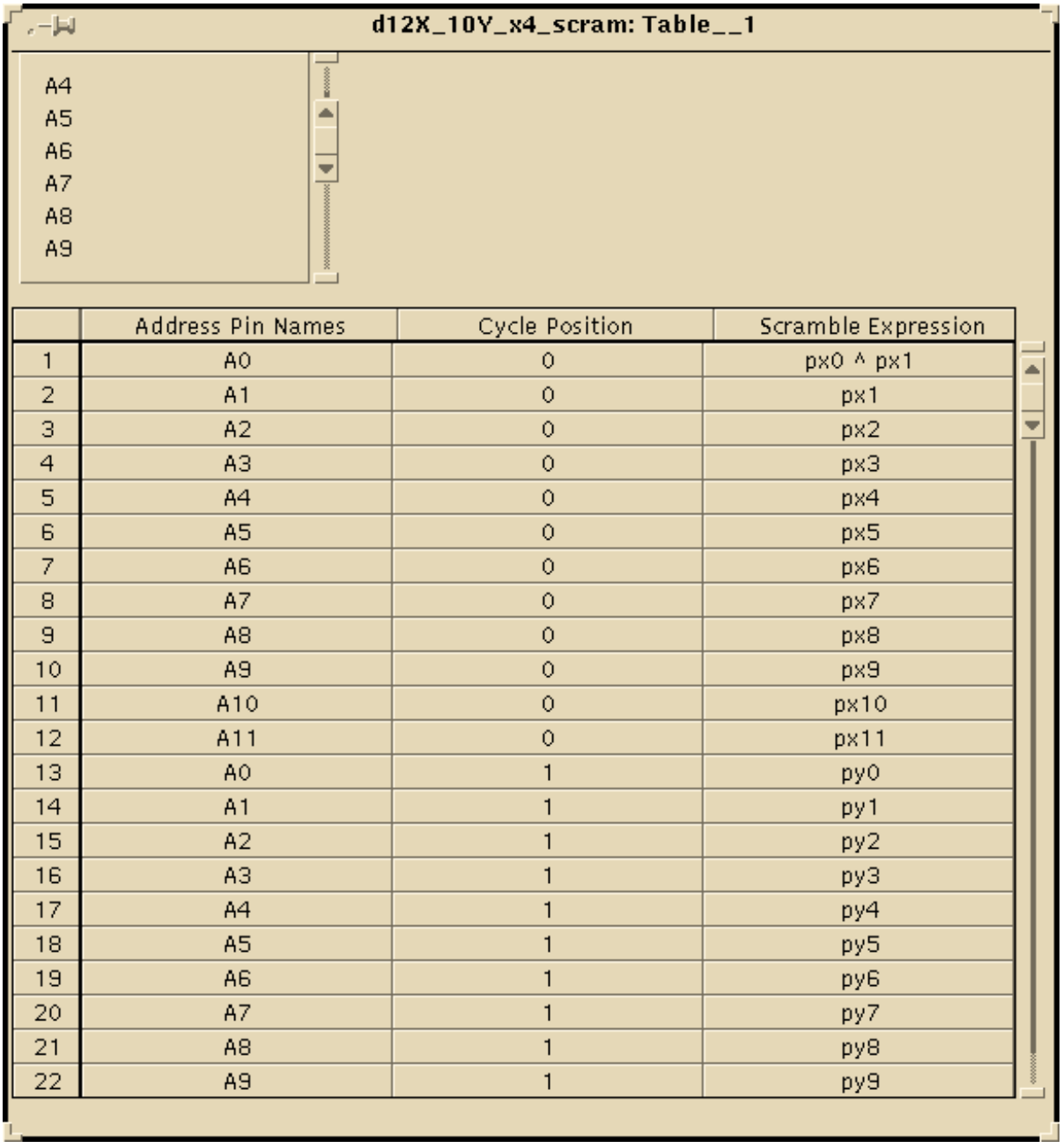
Note: Non-address-multiplexed RAMs specify a suffix parameter value of *0* for all pins specified in the *addr-pins* list. In the example above, the mutadr equivalent to the first table entry is:

```
(A0$0     "px0^py1")
```

The mutadr equivalent to the last table entry is:

```
(A9$1     "py9")
```

The figure on the next page shows the Insted display for the instance shown above. It shows in a graphic format how the addr_pins, the value suffix, and the scram_expressions are related.



	Address Pin Names	Cycle Position	Scramble Expression
1	A0	0	px0 ^ px1
2	A1	0	px1
3	A2	0	px2
4	A3	0	px3
5	A4	0	px4
6	A5	0	px5
7	A6	0	px6
8	A7	0	px7
9	A8	0	px8
10	A9	0	px9
11	A10	0	px10
12	A11	0	px11
13	A0	1	py0
14	A1	1	py1
15	A2	1	py2
16	A3	1	py3
17	A4	1	py4
18	A5	1	py5
19	A6	1	py6
20	A7	1	py7
21	A8	1	py8
22	A9	1	py9

Figure 4.2-1: Insted Table Display of the d12X_10Y_x8_scram Instance

MFP Address Scramble Template 2

MFP Address Scramble Template 2 (Insted name: *mfp_addr_scram_2_Template*) contains address scramble information for the memory-under-test. This template is based on address mapping functions included in the user's job and are appropriate for devices where the number of rows, columns, or IOs is not an even power of two.

Parameters The MFP Address Scramble Template 2 has the following parameters.

Table 4.2-2. MFP Address Scramble Template 2 Parameters

Insted Prompt	Parameter	Data Type	Entry Required	Description
Which Patgen?	patgen	Integer	No (defaulted)	Specifies which patgen (MPG/scrambler) will be used. Choices are Main, Alt, and Both. Default = Both.
User function	addr_func	C function	No*	Name of the user function which maps patgen x,y to pin x,y. Default is 1-to-1 scrambling. * These functions must be included by the user in one of the device program's .t files.[a]
X scramble RAM h,v bits	x_in	list of strings	No	Patgen x, y bits to connect into the X scramble RAM.
Y scramble RAM h,v bits	y_in	list of strings	No	Patgen x, y bits to connect into the Y scramble RAM.
Input to bypass lines	unscram_in	list of strings	No	Patgen x, y bits to connect into bypass lines.
Output of x scramble RAM	x_out	list of strings	No	Pin x,y bits coming out from X scramble RAM.
Output of y scramble RAM	y_out	list of strings	No	Pin x,y bits coming out from Y scramble RAM.
Output from bypass lines	unscram_out	list of strings	No	Pin x, y bits coming out from bypass lines.
X address bits	pin_x_bits	list of strings	Yes	X address bits. LSB -> MSB
Y address bits	pin_y_bits	list of strings	Yes	Y address bits. LSB -> MSB

a. The user should wrap each such function in an extern C block. This means that the function definition is preceded by `extern "C"{` and followed by an extra closing }.

Example The following example instance shows programming for MFP Address Scramble Template 2.

mfp_addr_scram instance (mfp_addr_scram_2_Template)

```
(define-instance mfp_addr_scram  mfp_addr_scram_2_Template
   (value patgen 0 :choice "Main")
   (value addr_func mfp_addr_scram_func)
   (value x_in px0 px1 px2 px3 px4 px5 px6 px7)
   (value y_in py0 py1 py2 py3 py4 py5 py6)
   (value unscram_in)
   (value x_out A12 A7 A6 A5 A0 A10 A9 A11)
   (value y_out A8 A13 A14 A4 A3 A2 A1)
   (value unscram_out)
   (value pins_x_bits A12 A7 A6 A5 A0 A10 A9 A11)
   (value pins_y_bits A8 A13 A14 A4 A3 A2 A1)
)
```

Example The following example shows a user function that could be used to do one-to-one scrambling.

```
extern "C" {
int
mfp_addr_scram(TT_MFP_XYB*      unscram,
               TT_MFP_XYB*      scram)
{
               *scram = *unscram;

               return 0;
}
}
```

MFP Bitmap Usage Template

MFP Bitmap Usage Template (Insted name: *mfp_bitmap_usage_Template*) contains parameters that specify characteristics of bitmap information to be captured.

Parameters

MFP Bitmap Usage Template has the following parameters, listed **in order of appearance in Insted**.

Table 4.2-3. MFP Bitmap Usage Template Parameters

Insted Prompt	Parameter	Data Type	Entry Required	Description
Number of Sites to Bitmap in Parallel	num_sites	Integer	Yes	Specifies the number of sites that will be captured in parallel.
Store Mask Data	mask	Integer	No*	Specifies whether to reserve half of the Catch RAM for storing mask data. *__mask__ and __accum__ cannot both be On. Choices are 0 (Off), 1 (On).
Accumulate Bitmaps	accum	Integer	No*	Specifies whether to reserve half of the Catch RAM for storing accumulative bitmaps. *__mask__ and __accum__ cannot both be On. Choices are 0 (Off), 1 (On).

Example The following example instance shows programming for MFP Bitmap Usage Template.

d12X_10Y_x8_8duts_bitmap_usage instance (mfp_bitmap_usage_Template)

```
(define-instance  d12X_10Y_x8_8duts_bitmap_usage  mfp_bitmap_usage_Template
   (value num_sites 8)
   (value mask 0 :choice "Off")
   (value accum 0 :choice "Off")
)
```

MFP Dgen Template

MFP Dgen Template (Insted name: *mfp_dgen_Template*) is used to process and store instance data that contains data generator (dgen) connections and topological inversion information for the device.

The MFP Dgen Template Pin Table

MFP Dgen Template contains an Insted "table" that lists an equal number of

Data pin names
Cycle positions
Data generator specifications
Topological data inversion expressions

Each horizontal "line" in the table specifies dgen usage and inversion information for one cycle of each data pin listed in column 1 of the table.

NOTE **Each occurrence of an inversion expression for a particular dgen should match all other occurrences for that dgen**.

Parameters

MFP Dgen Template has the following parameters, listed **in order of appearance in Insted**.

Table 4.2-4. MFP Dgen Template Parameters

Insted Prompt	Parameter	Data Type	Entry Required	Description
Which Patgen?	patgen	Integer	No (defaulted)	Specifies which patgen (MPG/ scrambler) will be used. Choices are 0 (Main), 1 (Alt), and 2 (Both). Default = 0 (Main).
Data Pin Names	data_pins	Pinlist	Yes*	*Pin table data — column "1" fields* Lists pin names for which dgen and inversion information is provided. *For each **data_pins** entry, there must be corresponding entries in **suffix**, **dgen**, and **topo_inv_expr**.

Table 4.2-4. MFP Dgen Template Parameters (Continued)

Insted Prompt	Parameter	Data Type	Entry Required	Description
Cycle Position	suffix	List of integers	Yes*	*Pin table data — column "2" fields* **For IO pins**, specifies the connections to the dmux0 and dmux1 inputs. (dmux1 is generally used only for SCIO or mach2). **For drive-only pins,** specifies the connections for the X and Y halves of the channel. Choices are 0 (dmux0), 1 (dmux1). *For each **data_pins** entry, there must be corresponding entries in **suffix**, **dgen**, and **topo_inv_expr**.
Which Dgen?	dgen	List of integers	Yes*	*Pin table data — column "3" fields* Specifies the data generator connection for each pin in **data_pins**. *For each **data_pins** entry, there must be corresponding entries in **suffix**, **dgen**, and **topo_inv_expr**. Choices are 0 (dgen 0), 1 (dgen 1).
Invert Expression	topo_inv_expr	List of strings	Yes*	*Pin table data — column "4" fields* *For each **data_pins** entry, there must be corresponding entries in **suffix**, **dgen**, and **topo_inv_expr**. Contains quoted strings giving a topological inversion expression for each pin in **data_pins**. **topo_inv_expr** may be 0 to indciate "no inversion".

Example The following example instance shows programming for MFP Dgen Template.

d12X_10Y_x8_dgen instance (mfp_dgen_Template)

```
(define-instance  d12X_10Y_x8_dgen  mfp_dgen_Template
   (value value patgen 0: choice "Main")
   (value data_pins DQS)
   (value suffix 0)
   (value dgen 0)
   (value topo_inv_expr "px0")
)
```

MFP MUT Description Template

MFP MUT Description Template (Insted name: *mfp_mut_description_Template*) contains a description of the overall characteristics of the device, including test system speed, number of row and column addresses, number of device outputs, and a list of output pin names.

Parameters MFP MUT Description Template has the following parameters, listed **in order of appearance in Insted**.

Table 4.2-5. MFP MUT Description Template Parameters

Insted Prompt	Parameter	Data Type	Entry Required	Description
Tester Speed in MHz	speed	Integer	No (defaulted)*	Specifies the test system speed. Choices are: 60 (60MHz), 100 (100MHz) 120 (120MHz) 200 (200 MHz). *Default = 0 (60MHz) for J993, J994, and J995 test systems. This field is required and NOT defaulted for J997 test systems.
XY orientation	orientation	List of Integers	No	Specifies XY orientation; 0 (normal) or 1 (flipped)
Numer of X addresses	rows	Integer	No*	Specifies the number of X addresses. * Supported only for backwards compatibility
Number of Y addresses	cols	Integer	No*	Specifies the number of Y addresses. * Supported only for backwards compatibility.
Number of Output Bits	num_outputs	Integer	Yes	Specifies number of output bits listed in *Output Pin Names* list. Minimum = 1 (must be a power of 2).
Output Pin Names	output_pins	Pinlist	Yes	Lists output pin names.

Example The following example instance shows programming for MFP MUT Description Template.

d12X_10Y_x8_ios_60MHz_mut_desc instance (mfp_mut_description_Template)

```
(define-instance  d12X_10Y_x8_60MHz_mut_desc mfp_mut_description_Template
   (value speed 60 :choice "60MHz")
   (value max_h_pxpy_bit py9)
   (value max_v_pxpy_bit px11)
   (value num_outputs 8)
   (value output_pins DQ0 DQ1 DQ2 DQ3 DQ4 DQ5 DQ6 DQ7)
)
```

MFP MUT Topology Template

MFP MUT Topology Template (Insted name: *mfp_mut_topology_Template*) contains a description of the internal layout (topology) of the device. This template can be used to generate custom topologies that differ from the default topology MFP generates from the MUT description template.

The template is based on per-address-bit equations and is appropriate for devices where the number of rows, columns, and IOs are all even powers of two.

The MFP MUT Topology Bit Table

MFP MUT Topology Template contains an Insted "table" that lists an equal number of

Patgen X/Y bits
H/V bits

These two lists must be the same length.

The next page contains several examples of the use of this template with different MUT topologies.

Parameters

MFP MUT Topology Template has the following parameters, listed **in order of appearance in Insted**.

Table 4.2-6. MFP MUT Topology Template Parameters

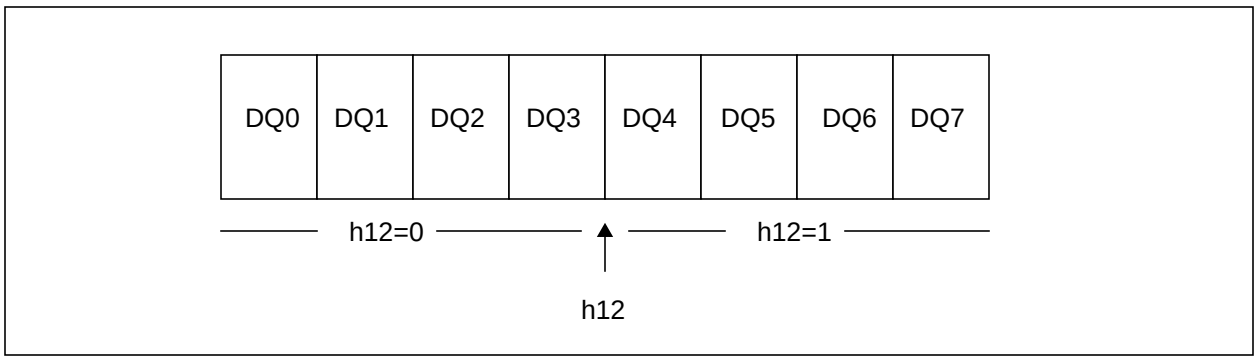
Insted Prompt	Parameter	Data Type	Entry Required	Description
Which Patgen?	patgen	Integer	No (defaulted)	Specifies which patgen (MPG/ scrambler) will be used. Choices are 0 (Main), 1 (Alt), and 2 (Both). Default = 2(Both).
Patgen X/Y Bit	x_y_bit	List of strings	Yes*	*Bit table data — column "1" fields* Lists patgen X/Y bits and I/O select bits. *For each **x_y_bit** entry, there must be a corresponding entry in **h_v_expr**.
Expression of H/V Bits	h_v_expr	List of strings	Yes*	*Bit table data — column "2" fields* Lists expressions of H/V bit determining values for the corresponding patgen X/Y or I/O select bits. H/V bits are from the horizontal and vertical addresses on the bitmap display. *For each **x_y_bit** entry, there must be a corresponding entry in **h_v_expr**.

Example The following example instances show programming for MFP MUT Topology Template. The topologies are included as comments in the beginning of each instance.

d12X_10Y_x8_topo_block instance (mfp_mut_topology_Template)

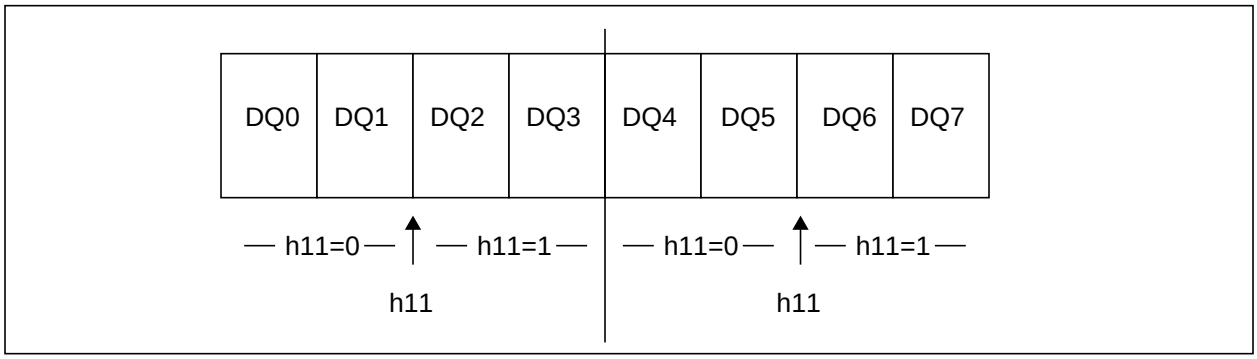
```
(define-instance  d12X_10Y_x8_topo_block  mfp_mut_topology_Template
;
;                                H (Y)
;
;         0-------1k-0-------1k-0------1k-0------1k-1k-------0-1k-------0-1k------0-1k------0
;          ____________________________________________________________________________________
;       0 |          |          |          |          |          |          |          |          |
;         .|          |          |          |          |          |          |          |          |
;         .|          |          |          |          |          |          |          |          |
;  V (X)  .|   DQ 0   |   DQ 1   |   DQ 2   |   DQ 3   |   DQ 4   |   DQ 5   |   DQ 6   |   DQ 7   |
;         .|          |          |          |          |          |          |          |          |
;         .|          |          |          |          |          |          |          |          |
;       4k |__________|__________|__________|__________|__________|__________|__________|__________|

   (value patgen 0 :choice "Main")
   (value x_y_bit
      "px0"  "px1"  "px2" "px3" "px4" "px5" "px6" "px7" "px8" "px9"
      "px10" "px11"
      "py0"  "py1"  "py2" "py3" "py4" "py5" "py6" "py7" "py8" "py9"
      "iob0" "iob1" "iob2"
   )
   (value h_v_expr
      "v0" "v1" "v2" "v3" "v4" "v5" "v6" "v7" "v8" "v9" "v10" "v11"
      "h0^h12" "h1^h12" "h2^h12" "h3^h12" "h4^h12" "h5^h12" "h6^h12"
      "h7^h12" "h8^h12" "h9^h12"
      "h10" "h11" "h12"
   )
)
```

Note: The x_y_bit and the h_v_expr merit additional consideration. Note first that the h12 bit differentiates between the left half (DQ0– DQ3) and the right half (DQ4 - DQ5). This is shown in the figure below.

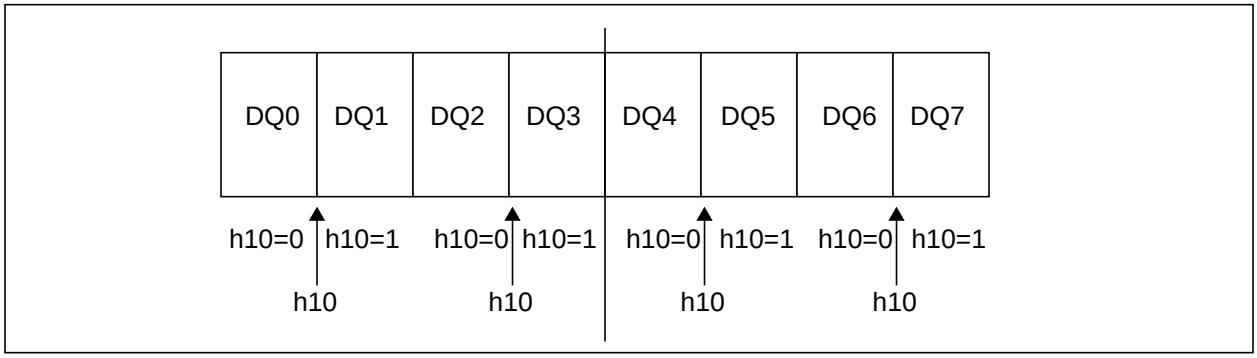


Likewise, the h11 bit differentiates between the left quarter (DQ0–DQ1) and the right quarter (DQ2–DQ3). It will also differentiate between the left quarter and the

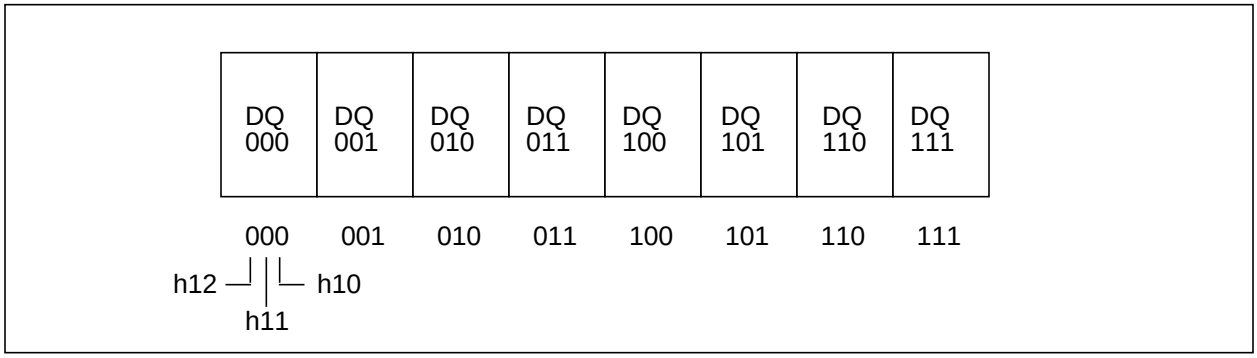
right quarter on the right half of the memory (where h12=1). This is shown in the figure below.



Finally the h10 bits differentiates between the left and right eighth of the memory. This is shown in the figure below



Now all three bits can be put together for all the IO regions:



These bits match the IO bits (iob0, iob1, and iob2). As shown above, h10 matches the least significant bit of the IO selects (iob0). Likewise, h11 matches iob1, and h12 matches iob2.

Finally, note that halfway through the memory, the addresses invert. That is, for DQ0 - DQ3, the addresses start with 0 and increase to 1k. For DQ4 - DQ7 the addresses start at 1k and decrease to 0. (Refer to the example on the previous page.) Since h12 differentiates between the left half and the right half. The h12 bit is exclusive-ORed with the other bits to obtain the inversion in the h_v_expr.

d12X_10Y_x8_topo_block_yx instance (mfp_mut_topology_Template)

```
(define-instance  d12X_10Y_x8_topo_block_yx  mfp_mut_topology_Template
;
;                                H (X)
;
;
;          0-------4k-0-------4k-0------4k-0------4k-4k-------0-4k-------0-4k------0-4k------0
;          ____________________________________________________________________________________
;        0|          |          |          |          |          |          |          |          |
;         .|          |          |          |          |          |          |          |          |
;         .|          |          |          |          |          |          |          |          |
;  V (X) .|   DQ 0   |   DQ 1   |   DQ 2   |   DQ 3   |   DQ 4   |   DQ 5   |   DQ 6   |   DQ 7   |
;         .|          |          |          |          |          |          |          |          |
;         .|          |          |          |          |          |          |          |          |
;       1k|__________|__________|__________|__________|__________|__________|__________|__________|

   (value patgen 0 :choice "Main")
   (value x_y_bit
      "px0"  "px1"  "px2" "px3" "px4" "px5" "px6" "px7" "px8" "px9"
      "px10" "px11"
      "py0"  "py1"  "py2" "py3" "py4" "py5" "py6" "py7" "py8" "py9"
      "iob0" "iob1" "iob2"
   )
   (value h_v_expr
      "h0^h14" "h1^h14" "h2^h14" "h3^h14" "h4^h14" "h5^h14" "h6^h14" "h7^h14"
      "h8^h14" "h9^h14" "h10^h14" "h11^h14"
      "v0" "v1" "v2" "v3" "v4" "v5" "v6" "v7" "v8" "v9"
      "h12" "h13" "h14"
   )
)
```

d12X_10Y_x8_topo_interleave instance (mfp_mut_topology_Template)

```
(define-instance  d12X_10Y_x8_topo_interleave  mfp_mut_topology_Template
;
;                              H (Y)
;
;      Y   0000000011111111.....................1K-1
;      DQ* 0123456701234567.....................7
;          ________________________________________
;        0|                                        |
;         .|                                        |
;         .|                                        |
;  V (X) .|                                        |
;         .|                                        |
;         .|                                        |
;       4k|________________________________________|
;
; * Note: "DQ" refers to the pseudo DQ as decoded by iob0 & iob1

   (value patgen 0 :choice "Main")
   (value x_y_bit
      "px0"  "px1"  "px2" "px3" "px4" "px5" "px6" "px7" "px8" "px9"
      "px10" "px11"
      "py0"  "py1"  "py2" "py3" "py4" "py5" "py6" "py7" "py8" "py9"
      "iob0" "iob1" "iob2"
   )
   (value h_v_expr
      "v0" "v1" "v2" "v3" "v4" "v5" "v6" "v7" "v8" "v9" "v10" "v11"
      "h3" "h4" "h5" "h6" "h7" "h8" "h9" "h10" "h11" "h12"
      "h0" "h1" "h2"
   )
)
```

MFP MUT Topology Template 2

MFP MUT Topology Template 2 (Insted name: *mfp_mut_topology_2_Template*) contains a description of the internal layout (topology) of the device.

This template can be used to generate custom topologies that differ from the default topology MFP generates from the MUT description template. The template is based on address mapping functions included in the user's job and is appropriate for devices where the number of rows, columns, or IOs is not an even power of two.

Table 4.2-7. MFP Memory Under Test Topology Template 2 Parameters

Insted Prompt	Parameter	Data Type	Entry Required	Description
Which Patgen?	patgen	Integer	No (defaulted)	Specifies which patgen (MPG/ scrambler) will be used. Choices are 0 (Main), 1 (Alt), and 2 (Both). Default = 2 (Both).
MUT width	width	positive integer	Yes	Width of the MUT. width = 2^(y bits)*(# ios)
MUT height	height	positive integer	Yes	Height of the MUT. height = 2^(xbits)
User function	addr_func	C function	No	Name of the user function which maps h,v to x,y,b (patgen x,y). * These functions must be included by the user in one of the device program's .t files.[a]
X scramble RAM hv bits	x_in	list of strings	No	H,V bits to connect into X scramble RAM.
Y scramble RAM hv bits	y_in	list of strings	No	H, V bits to connect into the Y scramble RAM.
Input to bypass lines	unscram_in	list of strings	No	Patgen x, y bits to connect into bypass lines.
Output of x scramble RAM	x_out	list of strings	No	Patgen x, y bits and I/O select bits to output from X scramble RAM.
Output of y scramble RAM	y_out	list of strings	No	Patgen x, y bits and I/O select bits to output from Y scramble RAM.
Output from bypass lines	unscram_out	list of strings	No	What address of the ADS bus muxes will receive information from the patgen bits.

a. The user should wrap each such function in an "extern C" block. This means that the function definition is preceded by `extern "C" {` and followed by an extra closing }.

Example The following example instance shows programming for MFP MUT Topology Template 2.

mfp_mut_topology_low_level instance (mfp_mut_topology_2_Template)

```
(define-instance mfp_mut_topology_low_level mfp_mut_topology_2_Template
   (value patgen 0 :choice "Main")
   (value width 1152)
   (value height 256)
   (value addr_func mfp_hv_2_xyb_func)
   (value x_in v0 v1 v2 v3 v4 v5 v6 v7)
   (value y_in h0 h1 h2 h3 h4 h5 h6 h7 h8 h9 h10)
   (value unscram_in)
   (value x_out px0 px1 px2 px3 px4 px5 px6 px7)
   (value y_out py0 py1 py2 py3 py4 py5 py6 iob0 iob1 iob2 iob3)
   (value unscram_out)
)
```

Example The following example shows a user function for a nine IO device where each IO is to be displayed side-by-side for a given Y address before proceeding to the next Y address.

```
      extern "C" {
      int
      mfp_topol (TT_MFP_HV*     hv,
                 TT_MFP_XYB*    xyb)
      {
                 xyb->x = hv->v;
                 xyb->b = hv->h % 9;
                 xyb->y = hv->h / 9;

                 return 0;
      }
      }
```

MFP Setup Template

MFP Setup Template (Insted name: *mfp_setup_Template*) lists all MFP instance references that describe device and hardware conditions for one capture setup.

Parameters MFP Setup Template has the following parameters, listed **in order of appearance in Insted**.

Table 4.2-8. MFP Setup Template Parameters

Insted Prompt	Parameter	Data Type	Entry Required	Description
MUT Description Instance	mut_ description_ instance	String	No*	Specifies the instance containing the MUT description. *Either **mfpadr_infile** must be specified, or there MUST be entries in each of the following fields: mut_description_instance addr_scram_instance dgen_instance bitmap_usage_instance mut_topology_instances
Capture/Both Scrambling Instance	addr_scram_ instance	String	No*	Specifies the instance containing capture and scan address scramble information. *If the scan_scram_instance parameter is used, this parameter specifies capture-only address scramble information. *Either **mfpadr_infile** must be specified, or there MUST be entries in each of the following fields: mut_description_instance addr_scram_instance dgen_instance bitmap_usage_instance mut_topology_instances

Table 4.2-8. MFP Setup Template Parameters (Continued)

Insted Prompt	Parameter	Data Type	Entry Required	Description
Scan-Only Scrambling Instance	scan_scram_ instance	String	No*	Specifies the instance containing scan-only scrambling information. *This parameter is usually left blank. When this parameter is used, the addr_scam_instance parameter specifies capture-only scrambling information. *Either **mfpadr_infile** must be specified, or there MUST be entries in each of the following fields: mut_description_instance addr_scram_instance dgen_instance bitmap_usage_instance mut_topology_instances
Data Generator Instance	dgen_instance	String	No*	Specifies the instance containing data generator connections and topological inversion information. *Either **mfpadr_infile** must be specified, or there MUST be entries in each of the following fields: mut_description_instance addr_scram_instance dgen_instance bitmap_usage_instance mut_topology_instances
Bitmap Usage Instance	bitmap_usage_ instance	String	No*	Specifies the instance that describes characteristics of the bitmap information to be captured. *Either **mfpadr_infile** must be specified, or there MUST be entries in each of the following fields: mut_description_instance addr_scram_instance dgen_instance bitmap_usage_instance mut_topology_instances

Table 4.2-8. MFP Setup Template Parameters (Continued)

Insted Prompt	Parameter	Data Type	Entry Required	Description
MUT Topology Instance(s)	mut_topology_ instances	List of strings	No*	Specifies the instance(s) that describe MUT topology. *Either **mfpadr_infile** must be specified, or there MUST be entries in each of the following fields: mut_description_instance addr_scram_instance dgen_instance bitmap_usage_instance mut_topology_instances
Hardware Setup Output File	mfpadr_outfile	String	No	Specifies the name of the hardware setup output file.
Hardware Setup Input File	mfpadr_infile	String	No*	Specifies the name of the hardware setup input file. *Either **mfpadr_infile** must be specified, or there MUST be entries in each of the following fields: mut_description_instance addr_scram_instance dgen_instance bitmap_usage_instance mut_topology_instances

Example The following example instance shows programming for MFP Setup Template.

d12X_10Y_x8_j997_60MHz_mfp_setup instance (mfp_setup_Template)

```
(define-instance  d12X_10Y_x8_j997_60MHz_mfp_setup  mfp_setup_Template
   (value mut_description_instance  d12X_10Y_x8_ios_01234567_60MHz_mut_desc)
   (value addr_scram_instance       d12X_10Y_x8_scram)
   (value dgen_instance             d12X_10Y_x8_dgen)
   (value bitmap_usage_instance     d12X_10Y_x8_8duts_bitmap_usage)
   (value mut_topology_instances    d12X_10Y_x8_topo_block
                                    d12X_10Y_x8_topo_block_yx
                                    d12X_10Y_x8_topo_interleave
   )
   (value mfpadr_outfile MFP_OUTPUT_FILE)
   (value mfpadr_infile )
)
```

Chapter 4.3: Memory Fail Processing (MFP) High-Level Programming

Introduction

The Bitmap program together with MFP template instances can be used to capture fail data in the Catch RAM and display the data in bitmaps.This approach was explained in Chapter 4.2. An alternative users may want to consider is to use a job-driven approach for capture and scan operations. This chapter will describe high-level function calls and provide examples of their usage.

Hldsets

High-level function calls often assert (send to hardware) several hldsets when they are called. Hldsets provide grouping within the .mfpadr file (and the .mfpdb database) of related hardware areas. For instance, in the setups generated from template input, one hldset contains scramble RAM contents, multiplexer settings, and ADS assignments, since all of these things are interrelated. Hldsets appear in the MFP ADR database, and can be auto-generated from template instances, created by modifying auto-generated ADR databases, or created completely from scratch.

ADR Files

When the template input or high level function calls are not flexible enough to meet the user's needs, a custom ADR file can be used.

Generally, when you need to create a custom ADR file, start with a MFP Setup template and request that an ADR file be generated based on values in those instances. The generated ADR file can then be used as a starting point and further modified as needed. To get the starter ADR file written out, specify an ADR output file in the MFP Setup instance. Note that this setup generation happens during job run, at the first point where an MFP driver function is called which references that MFP Setup instance, so you need an active job to make this happen.

There are two ways that a hand-tuned (or created from scratch) ADR file may be used: with high-level functions, or with low-level driver calls.

The simplest case is to use the high-level calls just as you would use for unmodified, template-generated setups. In this approach, you start with template instances and specify an ADR output file. The generated ADR file is modified as

necessary, and then it is specified in the MFP Setup instance as an ADR input file. When a value is given for the ADR input file parameter, the MFP high-level drivers will bypass the normal setup-generation phase and use the contents of that ADR file. In this case, it is important that the edits made to the ADR file do not include modification of hldset names. The high-level drivers determine which hldsets to send to hardware based on operation type and hldset name. In general, the high-level drivers will assert several hldsets for a single action.

The second, and more difficult, approach is to used the custom ADR file in conjunction with the low-level driver calls. With this approach, you specify in each driver call the name of the hldset and a bitmask of one or more form-types (hardware subsystems) which should be sent from that hldset to hardware. In this way you can choose to create large hldsets and assert only a portion of them in a given driver call, or you can create small hldsets and assert them in their entirety, perhaps making multiple driver calls to assert the multiple hldsets to accomplish a single operation such as setting up for capture.

High Level Programming

For job-driven bitmapping, the high-level driver functions are most appropriate. In conjunction with template instances and the MFP databases automatically derived from the instances, these functions take care of most of the details involved in the creation of useful bitmaps. The high-level driver functions are:

High-Level Drivers	*Description*
`tt_mfp_map_pseudo_sites`	maps the active sites onto MFP pseudo sites
`tt_mfp_do_clear`	clears the Catch RAM (all or portion based on arguments)
`tt_mfp_func_test_setup`	set hardware up in preparation for simple functional testing
`tt_mfp_capture_setup`	set hardware up in preparation for capture
`tt_mfp_capture_complete`	inform MFP system that capture is finished
`tt_mfp_scan_setup`	set hardware up in preparation for scanning
`tt_mfp_do_scan`	scan from the Catch RAM to the Display RAM
`tt_mfp_scan_completed`	Inform MFP system that all scans for the previous capture are complete.
`tt_mfp_create_crik`	creates a custom CRIK (Catch RAM Index Key) map
`tt_mfp_destroy_crik`	removes a CRIK (Catch RAM Index Key) map
`tt_mfp_get_ra_error_count`	fetches redundancy analyzer error counts
`tt_mfp_get_ra_region_count`	fetches redundancy analyzer region counts
`tt_mfp_set_ra_all_rik`	sets redundancy analyzer's *all* RIK for specified head
`tt_mfp_set_ra_cycle_count`	sets redundancy analyzer cycle counts

High-Level Drivers	Description
tt_mfp_set_ra_error_count	sets the redundancy analyzer error count
tt_mfp_set_ra_region_count	sets the region count for specified redundancy analyzers
tt_mfp_set_ra_start_cycle	sets the redundancy analyzer start cycle value
tt_mfp_set_ra_stop_cycle	sets the redundancy analyzer stop cycle value
tt_mfp_set_es	Registers the edgeset label to modify the period of when scanning or clearing the Catch RAM
tt_mfp_get_es	
tt_mfp_set_waveform	Registers the waveform to select during scanning or clearing of the Catch RAM
tt_mfp_get_waveform	
tt_mfp_file_checks_on	Causes MFP to check for out of date .mfpdb files and retranslate the file if necessary.
tt_mfp_file_checks_off	Causes MFP **not** to check for out of date .mfpdb files.

The high-level function calls assert hardware-load driver sets (hldsets) to accomplish many of their functions. These hldsets, part of the MFP databases (and thus subject to editing using ADR translators), can be programmed independently with either the high-level or low-level programming approach.

These functions are declared in the `tt_mfp_job.h` file.

Note: Devices for which the number of outputs is *not* a multiple of 2 (such as by-9 devices) cannot use the high-level programming approach. These devices should use the low-level programming approach described in *Chapter 4.4, Memory Fail Processing Low-Level Programming.*

Job-Driven Bitmapping Flow

The job-driven bitmapping approach is illustrated in the following flow:

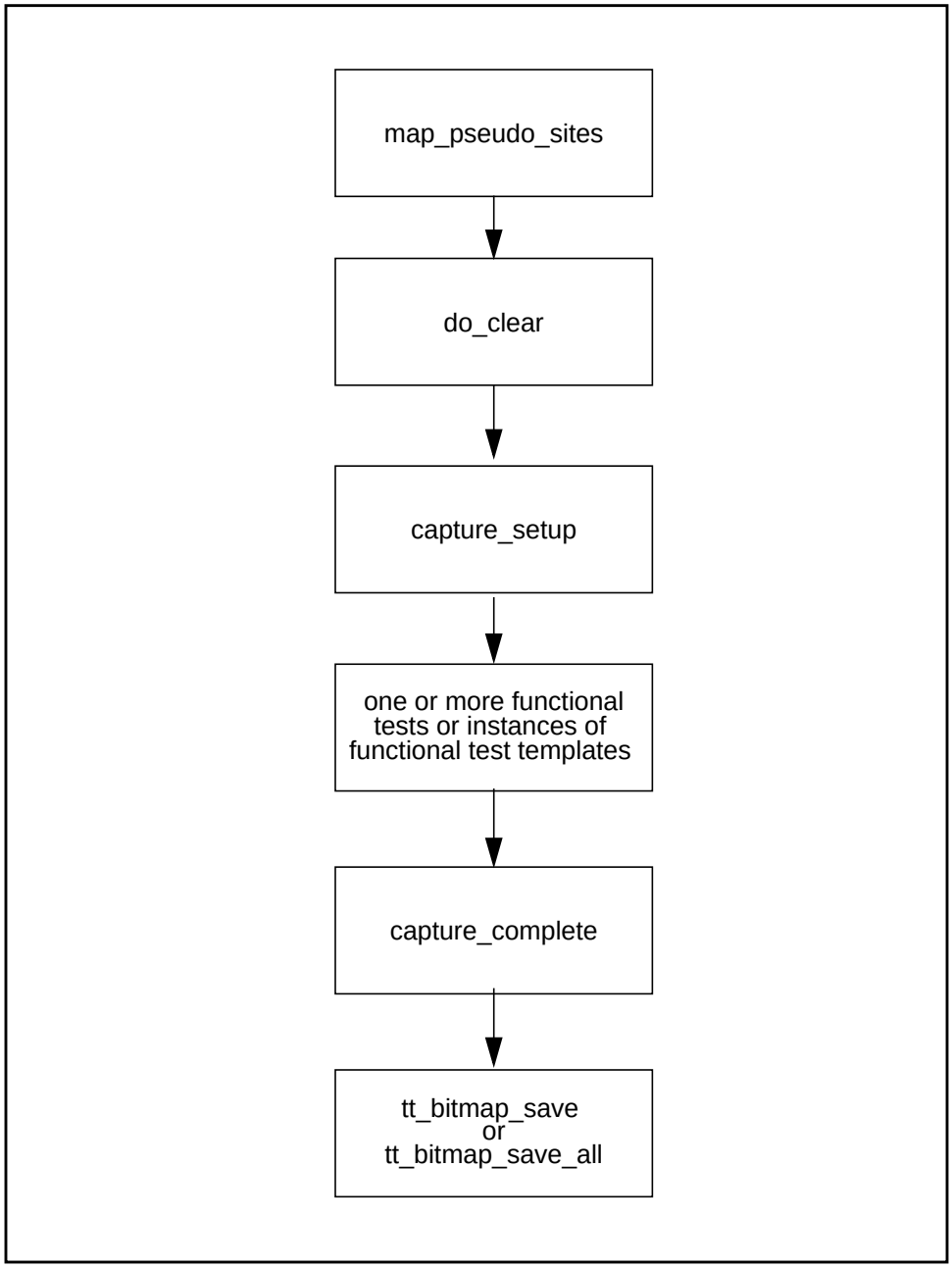


Figure 4.3-2: Job-Driven Bitmap Flow

Argument Types

A discussion of each function and its arguments follows. First some argument types (declared in tt_mfp_job.h and tt_mfp_types.h) used in many of the function calls need to be discussed.

TT_MFP_OP_TYPE

One argument type is TT_MFP_OP_TYPE. This is the operation type argument. The following are arguments of this type:

TT_MFP_OP_TYPE *Arguments*
TT_MFP_NULL_OP
TT_MFP_CURRENT
TT_MFP_ACCUM
TT_MFP_MASK
TT_MFP_MASKED
TT_MFP_UNMASKED

These TT_MFP_OP_TYPE arguments are passed to the MFP high-level driver functions. The meaning of some of these arguments vary depending on which function is being called. Accordingly, descriptions of these arguments will be included in the discussion of the high-level functions calls below.

TT_MFP_TOPO_INV

Another argument type is TT_MFP_TOPO_INV. This type has two arguments:

TT_MFP_TOPO_INV *Arguments*
TT_MFP_TOPO_INV_OFF
TT_MFP_TOPO_INV_ON

The TT_MFP_TOPO_INV type indicates if capture should be done with topological data inversion enabled or disabled. The topological data itself comes from the data generator template instance.

tt_mfp_map_pseudo_sites

The syntax of this function call is:

```
tt_mfp_map_pseudo_sites("mfp_setup_instance_name")
```

This function is called to map the physical (active sites) onto the pseudo sites that the MFP system deals with. The first N sites from the ORed list of active sites on each head are mapped to the N pseudo sites that can be set up by the specified MFP instance. The same set of sites is set up on each head. For example,

Head 0 Active Sites	*Head 1 Active Sites*	*ORed List of Active Sites*
1, 2, 5, 6	2, 3, 5, 7	1, 2, 3, 5, 6, 7

If the MFP setup allows for four active sites to be set up in parallel, the pseudo site mapping for these active sites would be:

Pseudo Site	*Physical Site (Both Heads)*
0	1
1	2
2	3
3	5

The MFP setup instance contains a field in which the desired MFP bitmap usage instance is specified; then, this bitmap usage instance contains a parameter (num_sites) in which the number of pseudo sites is specified. When to change this mapping is left to the programmer. However, if no mapping has been specified at do_clear or capture setup time, a mapping will be created for you.

The following is an example:

```
// map the first four active sites in the sitemask to Catch RAM
printf("MFP call: Mapping active sites to MFP pseudo sites\n");
tt_mfp_map_pseudo_sites("d12X_10Y_x8_mfp_setup");
```

In this example, the first four active sites in the sitemask are set up as Catch RAM pseudo sites. These first four sites are the only ones from which fail data is collected. The actual number of pseudo sites comes from the `d12X_10Y_x8_mfp_setup` instance which references the `d12X_10Y_x8_bitmap_usage` instance. The `d12X_10Y_x8_bitmap_usage` instance contains a parameter num_sites which is set to four.

tt_mfp_do_clear

The syntax of this function call is:

```
tt_mfp_do_clear("mpf_setup_instance_name",TT_MFP_OP_TYPE)
```

This function sets up hardware appropriately, selects a clear pattern, and then runs that pattern to clear the specified portion of the Catch RAM.

Possible values for the OP_TYPE arguments are:

OP_TYPE arguments for tt_mfp_do_clear	*Description*
`TT_MFP_NULL_OP` *or* `TT_MFP_CURRENT`	Either of these will set up to clear only the current results, even if masking or accumulation is enabled in the MFP setup instance.
`TT_MFP_MASK`	This will set up to clear both the current and mask regions of the Catch RAM.
`TT_MFP_ACCUM`	This will set up to clear both the current and accumulate regions of the Catch RAM.

The following is an example:

```
printf("MFP call: Clearing Catch RAM\n");
tt_mfp_do_clear("d12X_10Y_x8_mfp_setup",TT_MFP_NULL_OP);
```

After sending a message to the jobio window, this function call will clear the current results in the Catch RAM. All scramble RAM tables and multiplexers will be set up as specified in the d12X_10Y_x8_mfp_setup instance.

One hldsets is asserted when this function is called:

clear-both or *clear-lower* — one of these two hldsets will be asserted next. This sets up the appropriate clear-enable values on the Catch RAM modules to clear the specified regions. The ***clear-lowe***r hldset is used when you are set up to do masking or accumulating and the OP_TYPE argument is set to either TT_MFP_NULL_OP or TT_MFP_CURRENT. The ***clear-both*** hldset is asserted if you are not set up to do masking or accumulating, or if you are set up to do masking or accumulating and you select TT_MFP_MASK or TT_MFP_ACCUM. The choice of whether you will be doing masking, accumulate, or standard bitmapping is made in the mfp_bitmap_usage instance referenced by the setup instance called with this function call.

tt_mfp_func_test_setup

The syntax of this function call is:

```
tt_mfp_func_test_setup("mfp_setup_instance_name",
      TT_MFP_TOPO_INV, TT_MFP_OP_TYPE)
```

This function has the same functionality as `tt_mfp_capture_setup` except that it leaves the Catch RAM disabled and does not enforce any patgen mode checking in the start patgen driver. Basically, the scramble RAMs, multiplexers, topo invert RAM, and data generator are set up for functional testing.

It is recommended that this function call be used to set up scramble RAMs, multiplexers, and so on, even if you do not otherwise plan to use the Catch RAM.

tt_mfp_capture_setup

The syntax of this function call is as follows:

```
tt_mfp_capture_setup("mfp_setup_instance_name",
      TT_MFP_TOPO_INV, TT_MFP_OP_TYPE)
```

This function is called to setup for capture. Those hldsets which are part of that setup are asserted. (i.e. their software caches are sent to hardware). Before calling this function, you should call tt_mfp_do_clear with appropriate arguments.

Argument values and their meanings are as follows:

TT_MFP_ TOPO_INV:

TT_MFP_TOPO_INV_ON	Set up topological data inversion RAM and its input multiplexes according to the equations specified in the data generator template.
TT_MFP_TOPO_INV_OFF	Disable topological data inversion.

TT_MFP_OP_TYPE:

TT_MFP_ACCUM	Set up to capture into both the current and accumulative halves of the Catch RAM. This will be ineffective if you have not specified in your bitmap usage instance that you wanted to enable accumulate mode.
TT_MFP_MASK	Set up to capture the mask image for each currently enabled site. This will enable capturing into both the current and mask halves of the Catch RAM. You should call `tt_mfp_do_clear ("mfp_setup_instance_name", TT_MFP_MASK)` before making this call. You must have enabled the masking feature in the bitmap usage instance.
TT_MFP_CURRENT TT_MFP_NULL_OP	Either of these will set up to capture only to the current test portion of the Catch RAM. They are the only valid choices if your bitmap usage instance specifies neither masking nor accumulate. You should call `tt_mfp_do_clear ("mfp_setup_instance_name", TT_MFP_CURRENT)` before making this call.

The following is an example:

```
printf("MFP call: Catch RAM capture setup\n");
tt_mfp_capture_setup("d12X_10Y_x8_mfp_setup",TT_MFP_TOPO_INV_ON,
        TT_MFP_NULL_OP);
```

In this example, the `d12X_10Y_x8_mfp_setup` instance contains the information that will setup the scramble RAM, ADS bit assignments, and so on. The capture set up will occur, the topological-inversion RAM will be programmed, and because of the `TT_MFP_TOPP_INV_ON` argument, topological-inversion will be

active. The TT_MFP_NULL_OP argument means that only the current test portion of the Catch RAM will be set up for capture. If no mask or accumulate operations are being undertaken, all of the Catch RAM will be available for current test capture.

The function call treats the use of TT_MFP_NULL_OP identically to TT_MFP_CURRENT; however, the use of TT_MFP_NULL_OP in this function call suggests that no accumulate or mask operations are currently in use.

This high-level programming function call will assert five hardware-load driver sets:

cram-bg-disable hldset— This disables all Catch RAM slices so that only the slices explicitly enabled later will capture data.

capture-main hldset — This sets up scramble RAM image and multiplexers, ADS bit assignments (multiplexers), and channel multiplexers.

one of *topo-invert-on hldset* or *topo-invert-off hldset* — The choice is based on the value of the TT_MFP_ TOPO_INV argument. These hldsets program the topo-invert RAM and its input multiplexers according to the equations specified in the associated dgen template instance or programs all the multiplexers and the first RAM location to zeros.

clear-off hldsets — this programs the Catch RAM modules to have clear-enable disabled.

one of *capture-both hldset* or *capture-lower hldset* — This choice is based on the value of the TT_MFP_OP_TYPE. If the TT_MFP_OP_TYPE is TT_MFP_ACCUM or TT_MFP_MASK, the capture-both hldset will be selected. If the TT_MFP_OP_TYPE is programmed to any other value, the capture-lower hldset is used. These hldsets enable the appropriate Catch RAM slices and connect the right channels to the right slices.

tt_mfp_capture_completed

This syntax of this function call is:

```
tt_mfp_capture_completed()
```

There are no arguments for this high-level function

This function is called after the capture patterns have been run. It disables the Catch RAM slices and turns off patgen mode checking that occurs in start patgen after doing a setup for capture. When patgen mode checking is active, the current patgen mode is matched with the one specified in the MFP setup.

tt_mfp_scan_setup

The syntax of this function call is:

```
tt_mfp_scan_setup (head, pseudo_site,
       "mfp_mut_topology_instance_name", TT_MFP_OP_TYPE)
```

This function is called to set up for scanning from the Catch RAM to the Display RAM. Those hardware-load driver sets which are part of that setup are asserted. Generally, this function will be called for you by a call to tt_bitmap_save or tt_bitmap_save_all.

Argument values and their meanings are as follows:

head	This is the integer head number. It can be *0* or *1*.
pseudo_site	This is an integer in the range 0 to (num_sites -1) where num_sites is the number of devices to be tested in parallel. num_sites comes from the Bitmap Usage Template.
mfp_mut_topology_instance_name	This is the name of one of the instances of the MFP Memory-Under-Test Topology Template or it is *NULL*. If *NULL* is specified, an automatically generated default topology will be used. The default topology will display IOs side by side.

TT_MFP_OP_TYPE:

TT_MFP_CURRENT TT_MFP_UNMASKED TT_MFP_NULL_OP	Any of these will set up for the current (last captured) image to be scanned out.
TT_MFP_MASKED	If masking was enabled in the MFP setup instance, this will set up to scan out the current fails masked by the last saved mask image for that site. The bitmap display will only show new fails from the most recent capture which are not also present in the mask image.
TT_MFP_MASK	If masking was enabled in the MFP Setup instance, this will set up to scan out the mask image itself for the given site. Note the difference between this argument and the TT_MFP_MASKED argument.
TT_MFP_ACCUM	If accumulate was enabled in the MFP setup instance, this will set up to scan out the accumulated failures of all captures on this site since the last call to `tt_mfp_do_clear("mfp_setup_instance_name",TT_MFP_ACCUM)`

When this high-level function call is used, five hardware-load driver sets are asserted:

<topology_instance_name> hldset — This hldset's name matches the name of the topology instance specified as an argument in the `tt_mfp_scan_setup` function call. The topology instance defines the layout that will be used in the scan setup. If NULL is specified as the argument for the `tt_mfp_scan_setup` function, the hldset named *default-topology* will be used.

clear-off hldset — This programs the Catch RAM modules so that the clear-enable function is disabled (clear is turned off).

cram-bg-disable hldset — This disables all Catch RAM slices so that only the slices explicitly enabled later will be used during scanning.

scan-<ULB>-site<PSITE> hldset — This hldset enables the Catch RAM slices that should be scanned out. Which slices are enabled depends on which TT_MFP_OP_TYPE argument is passed in to the `tt_mfp_scan_setup` function.

- <ULB> is **upper** if the TT_MFP_OP_TYPE argument is TT_MFP_MASK or TT_MFP_ACCU
- <ULB> is **both** if the TT_MFP_OP_TYPE argument is TT_MFP_MASKED
- <ULB> is **lower** for all other cases
- <PSITE> is the pseudo site number which is being scanned out.

scan-disp-<MASK>-head<N> — This sets up the display processor hardware appropriately for the given TT_MFP_OP_TYPE arguments.

- <MASK> is **masked** or **unmasked**.
- N is **0** or **1** based on which head the scan data comes from.

tt_mfp_do_scan

The syntax of this statement is:

```
tt_mfp_do_scan()
```

This function performs the scan operation. In most cases this function is called for you by the tt_bitmap_save() function or the tt_bitmap_save_all function().

tt_mfp_scan_completed

This syntax of this function is:

```
tt_mfp_scan_completed()
```

This function is called after all scan operations that are to be done with the current capture data are completed. It will put the hardware back in a state where functional testing may be resumed.

tt_mfp_create_crik

This function call creates a custom CRIK (Catch RAM Index Key) map. The syntax of this function is:

```
tt_mfp_create_crik(slices_flex_array)
```

The *slices_flex_array* argument is the name of a flex array contain the CRIK mapping to be created. A flex array is an array whose first element represents the total number of elements contained in the array. In the case of the slice_flex_array, the remaining elements represent Catch RAM slice numbers.

An example follows:

```
Suppose that a CRIK is desired for slices {0, 4, 8, 12}, this
could be accomplished with the following code:

      int my_slice_flex_array[] = { 4, 0, 4, 8, 12 };
      crik_custom = tt_mfp_create_crik(my_slices_flex_array)
```

In this example, the first element in my_slice_flex_array is 4, indicating that four Catch RAM slice numbers will follow in the array.

tt_mfp_destroy_crik

This command removes a CRIK created using the tt_mfp_create_crik function from the list of available CRIKs.

The format of the function is

```
tt_mfp_destroy_crik(crik)
```

The *CRIK* argument is the name of a CRIK created using the tt_mfp_create_crik function call.

tt_mfp_get_ra_error_count

This function call fetches the redundancy analyzer error counts for the analyzers specified in the analyzer_list flex array. Results are returned in a flex array ra_error_count.

The format of the function call is:

```
tt_mfp_get_ra_error_count(ra_error_counts, analyzer_list, head)
```

The *ra_error_counts* argument is a flex array in which the error counts will be stored. This flex array must be equal in size to the analyzer_list (see example below).

The *analyzer_list* is the name of a flex array that contains a list of the redundancy analyzers to be used. This list of redundancy analyzers is sometimes referred to as a RIK or **r**edundancy analyzer **i**ndex **k**ey.

The *head* argument is used to specify which head is to have its errors read back. The head argument is an integer of value 0 or 1. Only one head can have its errors read back at a time.

An example follows:

```
//
// Read back region counts for analyzers 0,4,8,12,16 on head 0.
//
      int d1Mx4_analyzer_list[6] = { 5, 0, 4, 8, 12, 16 };
      int d1Mx4_ra_error_counts[6];
      int head = 0;

      tt_mfp_get_ra_error_count(d1Mx4_ra_error_counts,
          d1Mx4_analyzer_list, head);
```

tt_mfp_get_ra_region_count

This function call fetches the redundancy analyzer region counts for the analyzers specified in the analyzer_list flex array. Results are returned in a flex array ra_region_counts.

The format of the function call is:

```
tt_mfp_get_ra_region_count(ra_region_counts, analyzer_list,
        head)
```

The *ra_region_counts* argument is a flex array in which the error counts will be

stored. This flex array must be equal in size to the analyzer_list (see example for tt_mfp_get_ra_error_count above).

The *analyzer_list* is the name of a flex array that contains a list of the redundancy analyzers to be used. This list of redundancy analyzers is sometimes referred to as a RIK or **r**edundancy analyzer **i**ndex **k**ey.

The *head* argument is used to specify which head is to have its errors read back. The head argument is an integer of value 0 or 1. Only one head can have its errors read back at a time.

tt_mfp_set_ra_all_rik

This function sets the redundancy anayzers's *all* RIK for the specified head. A RIK is a **r**edundancy analyzer **i**ndex **k**ey. A RIK is a flex array used to specify redundancy analyzer numbers. The *all* RIK is the default RIK used by other MFP function calls.

The format of the function call is:

```
tt_mfp_set_ra_all_rik(analyzer_list, headmask)
```

The *analyzer_list* argument is a flex array of redundancy analyzer numbers. For more information on flex arrays see the tt_mfp_create_crik description on page 4.3.11.

The *headmask* argument specifies which head the function call applies to. The headmask argument may be:

- ❒ 1 - head 0 only (01)
- ❒ 2 - head 1 only (10)
- ❒ 3 - heads 0 and 1 (11)

The default headmask is 3, which will set the all start cycles the same for both heads. The headmask argument may be omitted.

tt_mfp_set_ra_cycle_count

This function call is used to set redundancy analyzer cycle counts.

The format of this function call is:

```
tt_mfp_set_ra_cycle_count(count,analyzer_list,headmask)
```

The *count* argument is the integer value the redundancy analyzer's cycle counter should be set to.

The *analyzer_list* is the name of a flex array that contains a list of the redundancy analyzers to be used. This list of redundancy analyzers is sometimes referred to as a RIK or **r**edundancy analyzer **i**ndex **k**ey. (For an example of how to create a flex array see the tt_mfp_create_crik discussion on page 4.3.11.) If `NULL` is specified instead of the name of analyzer list, then the current the current RA "all" RIK will be used (see the tt_mfp_set_ra_all_rik discussion on page 4.3.13.) Note that if an analyzer_list *is* specified, the RA "all" RIK will be reset to the value of the analyzer_list once this function call is executed. The analyzer_list defaults to NULL and may be omitted.

The *headmask* argument specifies which head the function call applies to. The headmask argument may be:

- ❒ 1 - head 0 only (01)
- ❒ 2 - head 1 only (10)
- ❒ 3 - heads 0 and 1 (11)

The default headmask is 3, which will set the all start cycles the same for both heads. The headmask argument may be omitted.

tt_mfp_set_ra_error_count

This function is used to set the redundancy analyzer set error count.

The syntax of the function call is:

```
tt_mfp_set_ra_error_count(count, analyzer_list, headmask)
```

The *count* argument is the integer value the redundancy analyzer's error counter should be set to.

The *analyzer_list* is the name of a flex array that contains a list of the redundancy analyzers to be used. This list of redundancy analyzers is sometimes referred to as a RIK or **r**edundancy analyzer **i**ndex **k**ey. (For an example of how to create a flex array see the tt_mfp_create_crik discussion on page 4.3.11.) If `NULL` is specified instead of the name of analyzer list, then the current the current RA "all" RIK will be used (see the tt_mfp_set_ra_all_rik discussion on page 4.3.13.) Note that if an analyzer_list *is* specified, the RA "all" RIK will be reset to the value of the analyzer_list once this function call is executed. The analyzer_list defaults to NULL and may be omitted.

The *headmask* argument specifies which head the function call applies to. The headmask argument may be:

- ❒ 1 - head 0 only (01)
- ❒ 2 - head 1 only (10)
- ❒ 3 - heads 0 and 1 (11)

The default headmask is 3, which will set the all start cycles the same for both heads. The headmask argument may be omitted.

tt_mfp_set_ra_region_count

This function call sets the region count for the specified redundancy analyzers.

The format of the function call is:

```
tt_mfp_set_ra_region_count(count, analyzer_list, headmask)
```

The *count* argument is the integer value the redundancy analyzer's region count should be set to.

The *analyzer_list* is the name of a flex array that contains a list of the redundancy analyzers to be used. This list of redundancy analyzers is sometimes referred to as a RIK or **r**edundancy analyzer **i**ndex **k**ey. (For an example of how to create a flex array see the tt_mfp_create_crik discussion on page 4.3.11.) If `NULL` is specified instead of the name of analyzer list, then the current the current RA "all" RIK will be used (see the tt_mfp_set_ra_all_rik discussion on page 4.3.13.) Note that if an analyzer_list *is* specified, the RA "all" RIK will be reset to the value of the analyzer_list once this function call is executed. The analyzer_list defaults to NULL

and may be omitted.

The *headmask* argument specifies which head the function call applies to. The headmask argument may be:

- ❒ 1 - head 0 only (01)
- ❒ 2 - head 1 only (10)
- ❒ 3 - heads 0 and 1 (11)

The default headmask is 3, which will set the all start cycles the same for both heads. The headmask argument may be omitted.

tt_mfp_set_ra_start_cycle

This function sets the RA start cycle value for all analyzers specified in the analyzer_list flex array.

The format for this function call is:

```
tt_mfp_set_ra_start_cycle(cycle, analyzer_list, headmask)
```

The *cycle* argument is the start cycle value.

The *analyzer_list* is the name of a flex array that contains a list of the redundancy analyzers to be used. This list of redundancy analyzers is sometimes referred to as a RIK or **r**edundancy analyzer **i**ndex **k**ey. (For an example of how to create a flex array see the tt_mfp_create_crik discussion on page 4.3.11.) If `NULL` is specified instead of the name of analyzer list, then the current the current RA "all" RIK will be used (see the tt_mfp_set_ra_all_rik discussion on page 4.3.13.) Note that if an analyzer_list *is* specified, the RA "all" RIK will be set to the value of the analyzer_list once this function call is executed. The analyzer_list defaults to NULL and may be omitted.

The *headmask* argument specifies which head the function call applies to. The headmask argument may be:

- ❒ 1 - head 0 only (01)
- ❒ 2 - head 1 only (10)
- ❒ 3 - heads 0 and 1 (11)

The default headmask is 3, which will set the all start cycles the same for both heads. The headmask argument may be omitted.

tt_mfp_set_ra_stop_cycle

This function sets the RA stop cycle value for all analyzers specified in the analyzer_list flex array.

The format for this function call is:

```
tt_mfp_set_ra_stop_cycle(cycle, analyzer_list, headmask)
```

The *cycle* argument is the stop cycle value.

The *analyzer_list* is the name of a flex array that contains a list of the redundancy analyzers to be used. This list of redundancy analyzers is sometimes referred to as a RIK or **r**edundancy analyzer **i**ndex **k**ey. (For an example of how to create a flex array see the tt_mfp_create_crik discussion on page 4.3.11.) If `NULL` is

specified instead of the name of analyzer list, then the current the current RA "all" RIK will be used (see the tt_mfp_set_ra_all_rik discussion on page 4.3.13.) Note that if an analyzer_list *is* specified, the RA "all" RIK will be set to the value of the analyzer_list once this function call is executed. The analyzer_list defaults to NULL and may be omitted.

The *headmask* argument specifies which head the function call applies to. The headmask argument may be:

- ❒ 1 - head 0 only (01)
- ❒ 2 - head 1 only (10)
- ❒ 3 - heads 0 and 1 (11)

The default headmask is 3, which will set the all start cycles the same for both heads. The headmask argument may be omitted.

tt_mfp_set_es
tt_mfp_get_es

These two statements register the edgeset label to modify the period of when scanning or clearing the FMM.

Example:

```
extern void
tt_mfp_es(Label* es);

extern Label*
tt_mfp_get_es();
```

tt_mfp_set_waveform
tt_mfp_get_waveform

These two statements register the waveform to select during scanning or clearing of the FMM. By default edgeset zero's period is modified as appropriate and then restored. If the user has registered an edgeset label, then that edgeset is modified instead.

Example:

```
extern void
tt_mfp_set_waveform(Waveform* wf);

extern Waveform*
tt_mfp_get_waveform();
```

tt_mfp_file_checks_on
tt_mfp_file_checks_off

By default MFP checks for out of date .mfpdb files and retranslates them if necessary. Since this check requires disk access it may hinder performance if using .mfpadr files. This checking can be toggled using the following functions:

```
extern void
tt_mfp_file_checks_on();

extern void
tt_mfp_file_checks_off();
```

Chapter 4.4: Memory Fail Processing (MFP) Low-Level Programming

Introduction

Memory Fail Processing software provides support for programming the error processing hardware for the purposes of bitmapping. The MFP software provides a complete low-level syntax and high-level support. This chapter describes the low-level approach.

Low-level Programming

When programming with the low-level MFP interface, you are provided with function interfaces for *capture setup* and *scan setup*; however, the actual scan and capture operations are left up to the programmer.

There are two versions of both the low-level capture setup and scan setup. One version takes a MFP setup instance name as an argument and the other version takes a mfpadr filename directly. If the low level is called with an MFP setup instance, then the input ADR file specified in that instance is extracted. This input ADR file is the only portion of the instance that will be used, all other fields are ignored. No high-level logic is ever applied when calling the low-level interface functions.

All hldsets that are associated with a given capture configuration should appear in the same ADR file. (That is, each DB file should contain one capture setup and may contain one or more related scan setups.) In this way all the software caches for the scan setups can be built using the same pseudo site to physical head/site mapping. The pseudo site part of these software caches will be built along with the capture setup portion which is why the related scan setups should be included with the capture setups.

Low-Level Function Calls

The following MFP low-level function calls are available:

Function Type	*Function Call*
Map Sites to Pseudo Sites function	tt_mfp_adr_map_pseudo_sites
Capture setup functions	tt_mfp_lowlevel_adr_capture_setup tt_mfp_lowlevel_capture_setup
Scan setup functions	tt_mfp_lowlevel_adr_scan_setup tt_mfp_lowlevel_scan_setup
Address Clock Channel Multiplexers	tt_mfp_set_addr_mux (two versions)
Address Scrambler RAM Input Multiplexers	tt_mfp_set_inmux
Address Scrambler RAM Output Multiplexers	tt_mfp_set_outmux
Clear Functions: Catch RAM Fail-Map Memory and Display RAM	tt_mfp_clear_catchram
	tt_mfp_clear_fmm
	tt_mfp_clear_display_ram
Display Processor Functions	tt_mfp_get_fail_cnt
	tt_mfp_get_saved_fc
	tt_mfp_save_fail_cnt
Disable Capture Functions for Catch RAM or FMM	tt_mfp_disable_catchram
	tt_mfp_disable_fmm
IO Channel Multiplexers	tt_mfp_set_data_mux (2 versions)
Scanning functions: Catch RAM or FMM to Display RAM	tt_mfp_scan_catchram_dr
	tt_mfp_scan_fmm_dr
Topological inversion function	tt_mfp_set_topo_block

tt_mfp_adr_map_pseudo_sites

The syntax of this function call is:

```
tt_mfp_adr_map_pseudo_sites("mfp_ADR_filename")
```

This function is called to map the physical (active) sites onto the pseudo sites that the MFP system deals with. The first N active sites are mapped to the N pseudo sites able to be set up for the specified ADR file. When to change this mapping is left up to the programmer. However, if no mapping has been specified at capture setup time, one will be created for you.

tt_mfp_lowlevel_adr_capture_setup
tt_mfp_lowlevel_capture_setup

The syntax of these function calls is as follows:

```
tt_mfp_lowlevel_adr_capture_setup("mfp_ADR_filename",
            "mfp_hldset_name", MFP_HLD_TYPE)

tt_mfp_lowlevel_capture_setup("mfp_setup_instance_name",
            "mfp_hldset_name", MFP_HLD_TYPE)
```

tt_mfp_lowlevel_adr_scan_setup
tt_mfp_lowlevel_scan_setup

The syntax of these function calls is as follows:

```
tt_mfp_lowlevel_adr_scan_setup("mfp_ADR_filename",
            "mfp_hldset_name", MFP_HLD_TYPE)

tt_mfp_lowlevel_scan_setup("mfp_setup_instance_name",
            "mfp_hldset_name", MFP_HLD_TYPE)
```

The arguments for these function calls are the same as for the capture setup calls described above. Refer to the description above.

TT_MFP_HLD_TYPE

The TT_MFP_HLD_TYPE argument defines which forms in the specified hldset ("mfp_hldset_name" argument) will be written to hardware. It is a bitmask that may be ORed together to write multiple hardware elements from an hldset with a single call. If TT_MFP_HLD_TYPE_ALL is selected for this argument, then all forms in an hldset will be written. An explanation of all the different form types in the descriptions below can be found later in this chapter.

TT_MFP_HLD_TYPE arguments	*Description*
TT_MFP_HLD_TYPE_SCRAM	inmux-form, outmux-form, x-scram-form, y-scram-form, addr-mux-form, data-mux-form
TT_MFP_HLD_TYPE_TOPO	topo-mux and topo-invert-form
TT_MFP_HLD_TYPE_MODULE	module-form
TT_MFP_HLD_TYPE_CATCHRAM	catchram-form
TT_MFP_HLD_TYPE_FAILMAP	failmap-form
TT_MFP_HLD_TYPE_DISPLAY_PROC	display-processor
TT_MFP_HLD_TYPE_RFL_MUX	rfl-mux-form
TT_MFP_HLD_TYPE_RFL_BUS	rfl-bus-form
TT_MFP_HLD_TYPE_RA_MUX	ra-mux-form

TT_MFP_HLD_TYPE arguments	*Description*
TT_MFP_HLD_TYPE_RA_FORM	ra-form
TT_MFP_HLD_TYPE_RA	ra
TT_MFP_HLD_TYPE_INMUX	inmux-form
TT_MFP_HLD_TYPE_OUTMUX	outmux-form
TT_MFP_HLD_TYPE_ADDR_MUX	addr-mux-form
TT_MFP_HLD_TYPE_DATA_MUX	data-mux-form
TT_MFP_HLD_TYPE_X_SCRAM	x-scram-form
TT_MFP_HLD_TYPE_Y_SCRAM	y-scram-form
TT_MFP_HLD_TYPE_TOPO_MUX	topo-mux
TT_MFP_HLD_TYPE_TOPO_INVERT	topo-invert-form
TT_MFP_HLD_TYPE_ALL	Everything contained in the hldset

tt_mfp_set_addr_mux

This function is used to select ADS input bits for each multiplexer on a drive-only (ACS) channel. There are two versions of this function call: one based on piks (pin-index keys) and the other based on pins. These function calls take advantage of the C++ function overloading feature: the correct function will be selected based on the arguments to the function call.

The syntax of the pik form is:

```
tt_mfp_set_addr_mux(pik,amux_0_bit,amux_1_bit, amux_0_bus)
```

The syntax of the pin form is:

```
tt_mfp_set_addr_mux(pin,amux_0_bit,amux_1_bit, amux_0_bus)
```

The first argument is either a pin or a pik. The *pik* argument is a pik number (integer). The *pin* argument is a pin name as defined in the pin database.

The *amux_o_bit* and *amux_1_bit* arguments may be an integer between 0 - 31 (inclusive). This will set the specified pik or pin to the specified ADS or SEC bit.

The *amux_0_bus* and *amux_1_bus* arguments may be TT_MFP_ADS or TT_MFP_SEC. Both bus arguments are defaulted to TT_MFP_ADS; they can be omitted if this selection is satisfactory.

tt_mfp_set_inmux

This function call is used to program the X and Y address scramblers RAM input multiplexers.

The syntax of this function call is:

tt_mfp_set_inmux(which, mux, pg_bit, bus)

The *which* argument is either TT_MFP_X_SCRAM or TT_MFP_Y_SCRAM. The *mux* argument is an integer of value 0 - 15 inclusive; it selects one of 16 X or 16 Y address scrambler multiplexers. The *pg_bit* is one of the TT_MFP_INMUX_XY types. The *bus* argument may be TT_MFP_ADS, TT_MFP_ADW, or TT_MFP_ADS|TT_MFP_ADW. The bus argument is defaulted to be TT_MFP_ADS|TT_MFP_ADW and may be omitted. If the TT_MFP_ADS|TT_MFP_ADW argument is selected, both ADS and ADW multiplexers will be programmed identically.

TT_MFP_INMUX_XY type

The TT_MFP_INMUX_XY type specifies the bits of the input multiplexers for the X and Y address scramblers. The arguments for the TT_MFP_INMUX_XY type are:

TT_MFP_INMUX_XY Arguments
TT_MFP_INMUX_ZERO
TT_MFP_INMUX_PX0
TT_MFP_INMUX_PX1
TT_MFP_INMUX_PX2
TT_MFP_INMUX_PX3
TT_MFP_INMUX_PX4
TT_MFP_INMUX_PX5
TT_MFP_INMUX_PX6
TT_MFP_INMUX_PX7
TT_MFP_INMUX_PX8
TT_MFP_INMUX_PX9
TT_MFP_INMUX_PX10
TT_MFP_INMUX_PX11
TT_MFP_INMUX_PX12
TT_MFP_INMUX_PX13
TT_MFP_INMUX_PX14
TT_MFP_INMUX_PX15
TT_MFP_INMUX_PY0
TT_MFP_INMUX_PY1
TT_MFP_INMUX_PY2
TT_MFP_INMUX_PY3
TT_MFP_INMUX_PY4
TT_MFP_INMUX_PY5
TT_MFP_INMUX_PY6
TT_MFP_INMUX_PY7
TT_MFP_INMUX_PY8
TT_MFP_INMUX_PY9
TT_MFP_INMUX_PY10
TT_MFP_INMUX_PY11
TT_MFP_INMUX_PY12
TT_MFP_INMUX_PY13
TT_MFP_INMUX_PY14
TT_MFP_INMUX_PY15

tt_mfp_set_outmux

This function call is used to program the address scramble RAM output multiplexers. (These multiplexers are sometimes referred to as the ADS input multiplexers.)

The syntax of this function call is:

```
tt_mfp_set_outmux(mux, scram_out_bit, bus)
```

The *mux* argument is an integer of value 0 - 31 inclusive; these are used to specify which output multiplexer is to be programmed. The *scram_out_bit* is one of the TT_MFP_OUTMUX_BITs. The *bus* argument may be TT_MFP_ADS, TT_MFP_ADW, or TT_MFP_ADS|TT_MFP_ADW. The bus argument is defaulted to be TT_MFP_ADS|TT_MFP_ADW and may be omitted. If the TT_MFP_ADS|TT_MFP_ADW argument is selected, both ADS and ADW multiplexers will be programmed identically.

TT_MFP_OUTMUX_BIT type

The TT_MFP_OUTMUX_BIT type is used to specify an address scrambler output multiplexer bit. (These bits are also known as the ADS input multiplexer bits.) The arguments for the TT_MFP_OUTMUX_BIT type are:

TT_MFP_OUTMUX_BIT Arguments
TT_MFP_OUTMUX_ZERO
TT_MFP_OUTMUX_SX0
TT_MFP_OUTMUX_SX1
TT_MFP_OUTMUX_SX2
TT_MFP_OUTMUX_SX3
TT_MFP_OUTMUX_SX4
TT_MFP_OUTMUX_SX5
TT_MFP_OUTMUX_SX6
TT_MFP_OUTMUX_SX7
TT_MFP_OUTMUX_SX8
TT_MFP_OUTMUX_SX9
TT_MFP_OUTMUX_SX10
TT_MFP_OUTMUX_SX11
TT_MFP_OUTMUX_SX12
TT_MFP_OUTMUX_SX13
TT_MFP_OUTMUX_SX14
TT_MFP_OUTMUX_SX15
TT_MFP_OUTMUX_SY0
TT_MFP_OUTMUX_SY1
TT_MFP_OUTMUX_SY2
TT_MFP_OUTMUX_SY3

TT_MFP_OUTMUX_BIT Arguments
TT_MFP_OUTMUX_SY4
TT_MFP_OUTMUX_SY5
TT_MFP_OUTMUX_SY6
TT_MFP_OUTMUX_SY7
TT_MFP_OUTMUX_SY8
TT_MFP_OUTMUX_SY9
TT_MFP_OUTMUX_SY10
TT_MFP_OUTMUX_SY11
TT_MFP_OUTMUX_SY12
TT_MFP_OUTMUX_SY13
TT_MFP_OUTMUX_SY14
TT_MFP_OUTMUX_SY15
TT_MFP_OUTMUX_DGEN0
TT_MFP_OUTMUX_DGEN1
TT_MFP_OUTMUX_ALT_DGEN0
TT_MFP_OUTMUX_ALT_DGEN1

tt_mfp_clear_catchram

This function call clears the Catch RAM. This syntax of this function is:

```
tt_mfp_clear_catchram(which)
```

The *which* argument is one of the TT_MFP_CRAM_CLEAR_SELECTIONS described below. This command defaults to TT_MFP_CRAM_CLEAR_BOTH which clears both halves of the Catch RAM. If this is the desired outcome, no argument need be specified.

TT_MFP_CRAM_CLEAR_SELECTION

The TT_MFP_CRAM_CLEAR_SELECTION type is used to specify which part of Catch RAM memory is to be cleared. This type has three arguments:

TT_MFP_CRAM_CLEAR_SELECTION *Arguments*
TT_MFP_CRAM_CLEAR_UPPER
TT_MFP_CRAM_CLEAR_LOWER
TT_MFP_CRAM_CLEAR_BOTH

tt_mfp_clear_fmm

This function call clears the fail map memory. This syntax of this function is:

```
tt_mfp_clear_fmm(count)
```

The *count* argument is the integer number of fail map memory locations to clear. Count is defaulted to 1 M (the entire fail map memory). If it is desired to clear the entire fail map memory, no argument is needed.

tt_mfp_clear_display_ram

This function call clears the display RAM. This syntax of this function is:

```
tt_mfp_clear_display_ram(count)
```

The *count* argument is the integer number of display RAM memory locations to clear. Count is defaulted to 4 M (the entire display RAM). If it is desired to clear the entire display RAM, no argument is needed.

tt_mfp_get_fail_cnt

This command reads the display processor fail counter. The syntax for this function is:

```
tt_mfp_get_fail_cnt()
```

The hardware value returned (an integer) is only valid until a waveform is restored or the period value is changed. If MFP function calls are used to do the scanning, the function tt_mfp_get_saved_fc(), described below, should be used since the hardware value may no longer be valid.

tt_mfp_get_saved_fc

This command reads back the fail count that was saved when the tt_mfp_save_fail_cnt() function, described below, was called. The function returns an integer.

The syntax for this function is:

```
tt_mfp_get_saved_fc()
```

tt_mfp_save_fail_cnt

This function call is used to save display processor hardware fail counts for use by bitmap or other programs, before the hardware value becomes invalid.

The syntax of this function is:

```
tt_mfp_save_fail_cnt(fc)
```

The *fc* argument is an integer fail count. This function call is of type void.

If the user is doing bitmap interposing for scan, the fail count should be read from hardware with the tt_mfp_get_fail_cnt command, and then the value obtained should be saved with tt_mfp_save_fail_cnt. This will make the value available for use by Bitmap in interactive mode.

tt_mfp_disable_catchram

This command disables capture on all Catch RAM slices.

The format for this function call is:

```
tt_mfp_disable_catchram()
```

tt_mfp_disable_fmm

This command disables FMM (fail-map memory) channels.

The format for this function is:

```
tt_mfp_disable_fmm()
```

tt_mfp_set_data_mux

This function is used to select ADS input bits for each multiplexer on a IO channel. There are two versions of this function call: one based on piks (pin-index keys) and the other based on pins. These function calls take advantage of the C++ function overloading feature: the correct function will be selected based on the arguments to the function call.

The syntax of the pik form is:

```
tt_mfp_set_data_mux(pik,dmux_0_bit,dmux_1_bit)
```

The syntax of the pin form is:

```
tt_mfp_set_data_mux(pin,dmux_0_bit,dmux_1_bit)
```

The first argument is either a pin or a pik. The *pik* argument is a pik number (integer). The *pin* argument is a pin name as defined in the pin database.

The *dmux_o_bit* and *dmux_1_bit* arguments may be an integer between 0 - 31 (inclusive). This will set the specified pik or pin for the specified ADS bit.

tt_mfp_scan_catchram_dr

This function scans a rectangular image of the device that has been captured in the Catch RAM to the display RAM. The scan pattern runs left to right, and top to bottom over the specified rectangle.

The format of this command is:

```
tt_mfp_scan_catchram_dr(rectp, do_clear)
```

The *rectp* argument is a pointer to a rectangular image of the device to scan. The rectp is a tt_mfp_rect type (see page 11).

The *do_clear* argument is zero, the display RAM will not be cleared prior to a scan. If do_clear is any other value, the display RAM will be cleared before scanning. The default is to clear the display RAM before scanning (non-zero). If the default condition is desired, this argument can be omitted.

tt_mfp_scan_fmm_dr

This function scans a rectangular image of the device that has been captured in the fail-map memory to the display RAM. The scan pattern runs left to right, and top to bottom over the specified rectangle.

The format of this command is:

```
tt_mfp_scan_fmm_dr(rectp, do_clear)
```

The *rectp* argument is a pointer to a rectangular image of the device to scan. The rectp is a tt_mfp_rect type.

The *do_clear* argument is zero, the display RAM will not be cleared prior to a scan. If do_clear is any other value, the display RAM will be cleared before scanning. The default is to clear the display RAM before scanning (non-zero). If the default condition is desired, this argument can be omitted.

tt_mfp_set_topo_block

The syntax of this function call is as follows:

```
/*
* Selects which block of the topological invert RAM will be selected for use.
* NOTE: High level MFP always programs and uses block 0.
* Bus is defaulted and may be omitted.
*/
extern int
tt_mfp_set_topo_block(int block,
int bus = TT_MFP_ADS | TT_MFP_ADW);
```

tt_mfp_rect

The tt_mfp_rect structure is defined as follows:

```
struct tt_mfp_rect {
int     horz;
int     vert;
int     height;
int     width;
};
```

Macro Definitions

In addition to the types described above, the following macros are also used with the MFP high-level function calls:

```
/*
 * Value returned if Display Processor fail counter overflows.
 */
#define TT_MFP_FAIL_CNT_OVERFLOW        0xfffffffd6
#define TT_MFP_ADS 1
#define TT_MFP_ADW2
#define TT_MFP_X_SCRAM 0
#define TT_MFP_Y_SCRAM 1
```

Setup Generation and Interactive Modifications

Software caches are generated on the first call to the low-level function calls in a given ADR file or instance. The software caches will be rebuilt if the ADR file changes or if the pseudo site mapping is changed.

In general, you may interactively change an MFP ADR file without reloading your job. The MFP software system will detect the fact that the .mfpadr file is newer than the corresponding .mfpdb file, reload, and rebuild the software caches.

mfpdb and mfpadr Translators

This section describes the low-level ASCII form of the MFP software system for the IG900+ software.

High Level/ Low Level

The memory fail processor software includes templates, a high level database generator, a set of hardware drivers, and translators to convert a database file to ASCII and ASCII to database. During normal high level operation, an ASCII file and database file are not required. The high level code builds an in-memory database and calls hardware drivers to load the hardware using the generated database.

If desired, you may request the high level database generator to dump an ASCII file. That file may be modified and used as an alternative to high level MFP when necessary. Alternatively, you may hand generate an ASCII input file.

MFP Translators

The translators are implemented as standard ADR900 style translators which will create a DB file from an ASCII file or vice-versa. The user may translate when off-line using ADR900.

When used on-line with MFP templates, the ASCII file may be specified as an input file and translation will be automatic when the ASCII file is newer than the database file.

The file extensions are .mfpdb for the database file and .mfpadr for the ASCII file.

Pin Names and Sites

The ASCII file contains pin names and pseudo sites. This allows the same ASCII file (and .mfpdb) to be used with different chansets for the same program, and the same ASCII file can be used on N or fewer real sites.

Hardware Initialization

Prior to setting up hardware, it is necessary to ensure that unprogrammed hardware (hardware not being used by this job) is not going to interfere with capture or scan operations. For example, if a Catch RAM slice is not used but its Scan Enable and Scan Compare registers are set to an arbitrary value then data from that unused slice could be ORed with data from a used slice during a scan operation. The MFPHLD drivers are used to do the initialization. The initialization is not automatic.

General ASCII File Layout

Inside an hldset statement all entries are optional (at least one is required). Inside any one of the sub statements, all entries are optional (at least one is required).

```
; comment line
(comment “file comment”)
(version “my version string”)
(use pinset “pin.pindb” pins)

(hldset name
   :set-type capture-main :comment “set 1” :capture-patgen-mode “100MHz”

   (default :bus ads)

   (inmux-form
        (:bus :mux :x   :y   )
        ( ads  0    px0  py0 )
   )

   (x-scram-form
        (:addr :data )
        ( 0     0xff )
   )

   (y-scram-form
        (:addr :data )
        ( 0     0xff )
   )

   (outmux-form
        (:mux :bit )
        ( 0    sx0 )
   )

   (addr-mux-form
        (:pin   :amux0-bus :amux1-bus :amux0 :amux1 )
        ((a0 a1) ads        sec        0      11    )
   )

   (data-mux-form
        (:pin :bus   :dmux0 :dmux1 )
        ( DQ0  ads0   4      1     )
   )

   (module-form
        (:module    :bus  :clear-enable )
        ((A0 A1 A2)  ads   off          )
   )

   (catchram-form
        (:array :by-mode :cs-enable :cs-compare :mode
             (:slice :scan-enable :scan-compare :pin :pseudo-site :or-pin ))
        ( 0      1        0xff       0x03        capture
             ( 0      0xff         0x03          DQ0  0            DQ2    ))
   )
```

```
    (failmap-form
     (:pin :pseudo-site :cs-enable :cs-compare :by-mode :mode
              :scan-enable :scan-compare)
     ( DQ0 (0 1)         0xff       0x03        1        capture
               0xff         0x03)
   )

   (display-processor :input fmm :enable on :head 0 :clear-enable on
      :horz-comp 2047 :vert-comp 1 :horz-width 2047 :fail-count-enable on
   )

   (topo-mux (px0 px1 px2 px3 py0 py1 py2 py3))

   (topo-invert-form
      (:block :addr :data )
      ( 0      0     3    )
   )

   (ra-mux-form
      (:board :mux :slice )
      ( A      0    0     )
      ( B      1    2     )
   )

   (ra-form
      (:analyzer :start :stop )
      ( 0         0      1023 )
   )

   (ra :or-mode or4)

   (rfl-mux-form
      (:module  :bus       :mux )
      ( A0       ads        0   )
      ( B0      (ads sec)   1   )
   )
   (rfl-bus-form
      (:board :upper-bus :lower-bus )
      ( A      ads        ads       )
      ( B      sec        ads       )
   )

)

(hldset name1  :set-type scan-main  :comment “set 2”
    . . . . .
)
```

Hldset Statement

There may be multiple hldset statements in one database but only one hldset may be of *:set-type* capture-main while there may be many *:set-type* scans.

The following is an example:

```
(hldset name :set-type capture-main :comment "set 1"
             :capture-patgen-mode "100MHz"
    (...)
)
```

Required fields include: hldset name. The other (optional) fields are:

- *:set-type* may be capture-main, capture-catchram, topo-invert, scan-main, clear, scan-catchram, or display-processor.
- *:comment* may be any quoted string.
- *:capture-patgen-mode* may be (quoted) "100 MHz", "60 MHz" or "50 MHz".

Default Statements

Default statements can precede any sub-statement within an hldset. If a default statement is not present and a hardware programming statement does not include a specific item to be programmed that hardware will not be touched.

One exception default is the *:bus* statement which designates a target piece of hardware. If none is mentioned then (ADS SEC) will be assumed.

inmux-form

The inmux-form is used to program the scramble RAM input multiplexers. Any multiplexers not programmed will be set to ZERO by default.

Example 1:

```
(inmux-form
   (:bus :mux :x   :y   )
   ( ads  0    px0  py0 )
   ( ads  1    px1  py1 )
)
```

Required fields include: *:mux* and at least one of *:x* or *:y*

- *:bus* may be ads, sec or (ads sec). (In this release (ads sec) are always programmed identically. :bus can be defaulted and is not required. If no bus is selected, the driver will default to (ads sec).
- *:mux* may be 0 to 15 for the *:x* mux selectors and 0 to 15 for the *:y* selectors. If a muxer is not mentioned in this form, it will be set to ZERO.
- *:x* and *:y* may be px0 through px15 or py0 through py15 or "ZERO".

Example 2 defaults the *:bus* (ads sec) is assumed by the driver.

```
(inmux-form
   (:mux  :x   :y   )
   ( 0     px0  py0 )
   ( 1     px1  py1 )
)
```

Example 3 forces the default *:bus* so both the ads and sec input multiplexers will be loaded with the same data.

```
(default :bus (ads sec))
(inmux-form
   (:mux  :x   :y   )
   ( 0     px0  py0 )
   ( 1     px1  py1 )
)
```

x-scram-form

The x-scram-form is used to program the contents of the x scramble RAM. Address followed by a stream of data. (16384 units of data 14 bits each plus address field range - for database design.)

The following is an example:

```
(x-scram-form
    (:bus  :addr  :data     )
    ( ads   1024  (0 1 2 3) )
)
```

Required fields include: *:addr* and *:data*

- *:bus* may be ads, sec or (ads sec). *:bus* can be defaulted and is not required. If no bus is selected, the driver will default to (ads sec).
- *:addr* may be 0 through 16,383 (may be a LIST or a RANGE).
- *:data* may be 0 through 16,383 (may be a LIST or a RANGE).

y-scram-form

The y-scram-form is used to program the contents of the y scramble RAM. Address followed by a stream of data.

The following is an example:

```
(y-scram-form
    (:bus :addr  :data         )
    ( ads  2048  (5 4 3 2 1 0) )
)
```

Required fields include: *:addr* and *:data*

- *:bus* may be ads, sec or (ads sec). *:bus* can be defaulted and is not required. If no bus is selected, the driver will default to (ads sec).
- *:addr* may be 0 through 65,535 (may be a LIST or a RANGE).
- *:data* may be 0 through 65,535 (may be a LIST or a RANGE).

outmux-form

The outmux-form is used to program the scramble RAM output multiplexers (also may be viewed as the ADS BUS input multiplexers). Any multiplexers not programmed will be set to ZERO by default.

The following is an example:

```
(outmux-form
    (:bus :mux :bit )
    ( ads  0    sx0 )
    ( ads  1    sy0 )
)
```

Required fields include: *:mux* and *:bit*

- *:bus* may be ads, sec or (ads sec). *:bus* can be defaulted and is not required. If no bus is selected, the driver will default to (ads sec).
- *:mux* may be 0 through 31. If a muxer is not mentioned in this form, it will be set to ZERO.
- *:bit* may be sx0 through sx15, sy0 through sy15, dgen0, dgen1, or "ZERO". The ADS bus only may have as an input alt-dgen0 or alt-dgen1.

addr-mux-form (mcs channel cards)

The addr-mux-form is used to program drive only channel connections to the ADS or SEC bus.

The following is an example:

```
(addr-mux-form
    (:pin :amux0-bus :amux1_bus :amux0 :amux1 )
    ( a0   ads        sec        0      11    )
)
```

Required fields include: *:pin* and at least one of *:amux0* or *:amux1*

- *:pin* may be any device pin name or pin group name found in the pinlist.
- *:amux0-bus* may be ads or sec. Can be defaulted and is not required. Driver defaults to ads.
- *:amux1-bus* may be ads or sec. Can be defaulted and is not required. Driver defaults to ads.
- *:amux0* may be a bus bit selection 0 through 31.
- *:amux1* may be a bus bit selection 0 through 31

data-mux-form (dcs channel cards)

The data-mux-form is used to program I/O channel/formatter connections to the ADS bus.

The following is an example:

```
(data-mux-form
    (:pin :bus   :dmux0  :dmux1 )
    ( DQ0  ads0   4       1     )
)
```

Required fields include: *:pin* and at least one of *:dmux0* or *:dmux1*

- *:pin* may be any device pin name or pin group name found in the pinlist.
- *:bus* may be ads0 or ads1. *:bus* can be defaulted and is not required. If no bus is selected, the driver will default to ads0.
- *:dmux0* may be a bus bit selection 0 through 31.
- *:dmux1* may be a bus bit selection 0 through 31.

module-form

The module-form statement is used to program catch RAM modules.

The following is an example:

```
(module-form
    (:module     :bus  :clear-enable )
    ((A0 A1 A2)   ads   off          )
    ((B0 A3 B1)   sec   on           )
)
```

Required fields include: *:module* and *:clear-enable*

- *:module* may be one of A0, A1, A2, A3, B0, B1, B2, B3. May be a LIST enclosed in parens (A2 B0 B3).
- *:bus* may be "ads" or "sec". *:bus* can be defaulted and is not required. If no bus is selected, the driver will default to ads.
- *:clear-enable* may be "on" or "off".

catchram-form

The catchram-form statement is used to program Catch RAM arrays and slices. It consists of an outer array and an inner slice section.

The following is an example:

```
(catchram-form
   (:array :by-mode  :cs-enable    :cs-compare    :mode
     (:slice   :scan-enable :scan-compare :pin :pseudo-site :or-pin))
   ((0..20) 1         0xff          0x03          capture
     ((0 1 2 3) 0x3f         0x01          DQ1  0           DQ2   ))
```

Required fields include: *:array, :slice* and at least one other field in each statement level.

- *:array* may be 0 through 39 (depends on system type and options installed). May be a LIST (1 3 5 7 9) or a RANGE (0..20) of numbers.
- *:by-mode* may be 1, 2, or 4. May be defaulted and is not required.
- *:cs-enable* may be 0 through 1023.
- *:cs-compare* may be 0 through 1023.
- *:mode* may be "capture", "capture-invert", "scan-disp", or "scan-ra". May be defaulted and is not required.
- *:slice* may be 0 through 3. May be a LIST (2 3) or a RANGE (0..3) of numbers.
- *:scan-enable* may be 0 through 1023.
- *:scan-compare* may be 0 through 1023.
- *:pin* may be any device pin name found in the pinlist. May be a single pin only. May be defaulted and is not required.
- *:pseudo-site* may be 0 through 31. May be a single site only. May be defaulted and is not required.
- *:or-pin* may be any device pin name found in the pinlist. May be a single pin only. May be defaulted and is not required.

failmap-form

The failmap-form statement is used to program the Fail Map Memory (FMM) hardware for bitmap capture. The following is an example:

```
(failmap-form
  (:pin :pseudo-site :cs-enable :cs-compare :mode :scan-enable :scan-compare)
  ( DQ0   (0 1)        0xff       0x03     capture     0xff         0x03)
 )
```

Required fields include :pin and at least one other field.

- :pin may be any device pin name found in the pinlist. It may be a single pin only. :pin may be defaulted and is not required.

  Pin "all" may be used to specify data for all test system channels in the failmap-form. This might be used to disable channels not used by the test program. Pin name "all" is reserved pin name in the IG900+ software system.

- :pseudo-site may be 0 through 31. It may be a single site only. :pseudo-site may be defaulted and is not required.
- :cs-enable may be 0 through 1023.
- :cs-compare may be 0 through1023.
- :mode may be "capture", "capture-invert", or "scan-disp". It may be defaulted and is not required.
- :scan-enable may be 0 through 1023.
- :scan-compare may be 0 through 1023.

display-processor

The display-processor statement is used to program the Display RAM controls.

The following is an example:

```
(display-processor :input fmm :enable on :clear-enable on
                   :horz-comp 2047 :vert-comp 1 :horz-width 2048
                   :fail-count-enable on
)
```

Required fields include: at least one field.

- *:input* may be fmm, cram-single, cram-0-mask, hi-cram-mask-lo, or lo-cram-mask-hi. Note: cram selections are J99x only.
- *:enable* may be “on” or “off”.
- *:clear-enable* may be “on” or “off”.
- *:horz-comp* may be 0 through 4095.

- *:vert-comp* may be 0 through 4095.
- *:horz-width* may be 0 through 2047.
- *:fail-count-enable* may be "on" or "off".

None of the display-processor attributes may be defaulted. All are optional.

topo-mux

The topo-mux statement selects which patgen addresses are assigned to the topological invert RAM.

The following is an example:

```
(topo-mux :bus ads :addr (px0 px1 py3 py4))
```

Up to 8 address lines may be assigned; MSB on the left, and LSB on the right.

Required fields include: *:addr* and its data

- *:bus* may be ads, sec or (ads sec). *:bus* can be defaulted and is not required. If no bus is selected, the driver will default to (ads sec)

topo-invert-form

The topo-invert-form statement is used to program the topological data inversion RAM.

The following is an example:

```
(topo-invert-form
    (:bus :block :addr :data )
    ( ads  0      0     3    )
    ( ads  0      1     0    )
)
```

Required fields include: *:addr* and *:data*

- *:bus* may be ads, sec or (ads sec). *:bus* can be defaulted and is not required. If no bus is selected, the driver will default to (ads sec).
- *:block* may be 0 through 3. *:block* may be defaulted. If left out, *:block* will be assumed to be 0.
- *:addr* may be 0 through 1023.
- *:data* may be 0 through 3 (0=no invert, 1=invert dgen0, 2=invert dgen1, 3= invert dgen0 and dgen1).

rfl-bus-form

The rfl-bus-form statement is used to program the region fail latch modules bus selection for the upper and lower region fail latch RAM.

The following is an example:

```
(rfl-bus-form
    (:board :upper-bus :lower-bus )
    ( A      ads        ads       )
    ( B      sec        ads       )
)
```

Required fields include: *:board* and at least one of *:upper-bus* or *:lower-bus*

- *:board* may be A or B.
- *:upper-bus* and lower-bus may be "ads", or "sec".

rfl-mux-form

The rfl-mux-form statement is used to program the region fail latch address bit selects for the A and B boards.

The following is an example:

```
(rfl-mux-form
    (:module  :bus      :mux )
    ( A0       ads       0   )
    ( B0      (ads sec)  1   )
)
```

Required fields include: *:module* and *:mux*

- *:module* may be one of A0, A1, A2, A3, B0, B1, B2, or B3.
- *:bus* may be ads, sec or (ads sec). *:bus* is optional and will default to (ads sec) by the driver.
- *:mux* may be 0 through 31, or ZERO.

ra-mux-form

The ra-mux-form statement programs slice to analyzer connections.

The following is an example:

```
(ra-mux-form
    (:mux :slice )
    ( 0    0     )
    ( 1    2     )
)
```

Required fields include: *:mux* and *:slice*

- *:mux* may be 0 through 71.
- *:slice* may be 0 through 38 or ZERO - not including 9, 19, or 29, which are hardware or'ed with the previous mux #. (8|9, 18|19, 28|29, 38|39)

ra-form

The ra-form statement is used to program the redundancy analyzer controls.

The following is an example:

```
(ra-form
  (:analyzer :start :stop :fault-addr :error-counter :cycle-counter )
  ( 0         0      1023  0           0              0             )
)
```

Required fields include: *:analyzer* and at least one of *:start*, *:stop*, *:fault-addr*, *:error-counter*, or *:cycle-counter*.

- *:analyzer* may be 0 through 35. May be a LIST (1 3 5 7) or a RANGE (0..35).
- *:start* may be 0 through 65,535. May be defaulted.
- *:stop* may be 0 through 65,535. May be defaulted.
- *:fault-addr* may be a value from 0 through 16,777,215.
- *:error-counter* may be a value from 0 through 4,294,967,295.
- *:cycle-counter* may be a value from 0 through 4,294,967,295.

ra :or-mode

The ra :or-mode statement is used to program analyzer or modes.

The following is an example:

```
(ra :or-mode or4)
```

Required fields include: *:or-mode*

- *:or-mode* may be “or2”, “or4”, “or10”, “or20”, or “orall”.

Chapter 5.1:Using Dual-MPG Mode

Introduction

As the speed of memory devices increases, it becomes necessary to take advantage of tester modes that allow increased pattern testing speeds. One such mode of operation is the "Dual Memory Pattern Generator" mode or "dual-MPG" mode. This mode allows the user to take advantage of a second Memory Pattern Generator (MPG) to generate device addresses and data at twice the tester's pattern generator frequency.

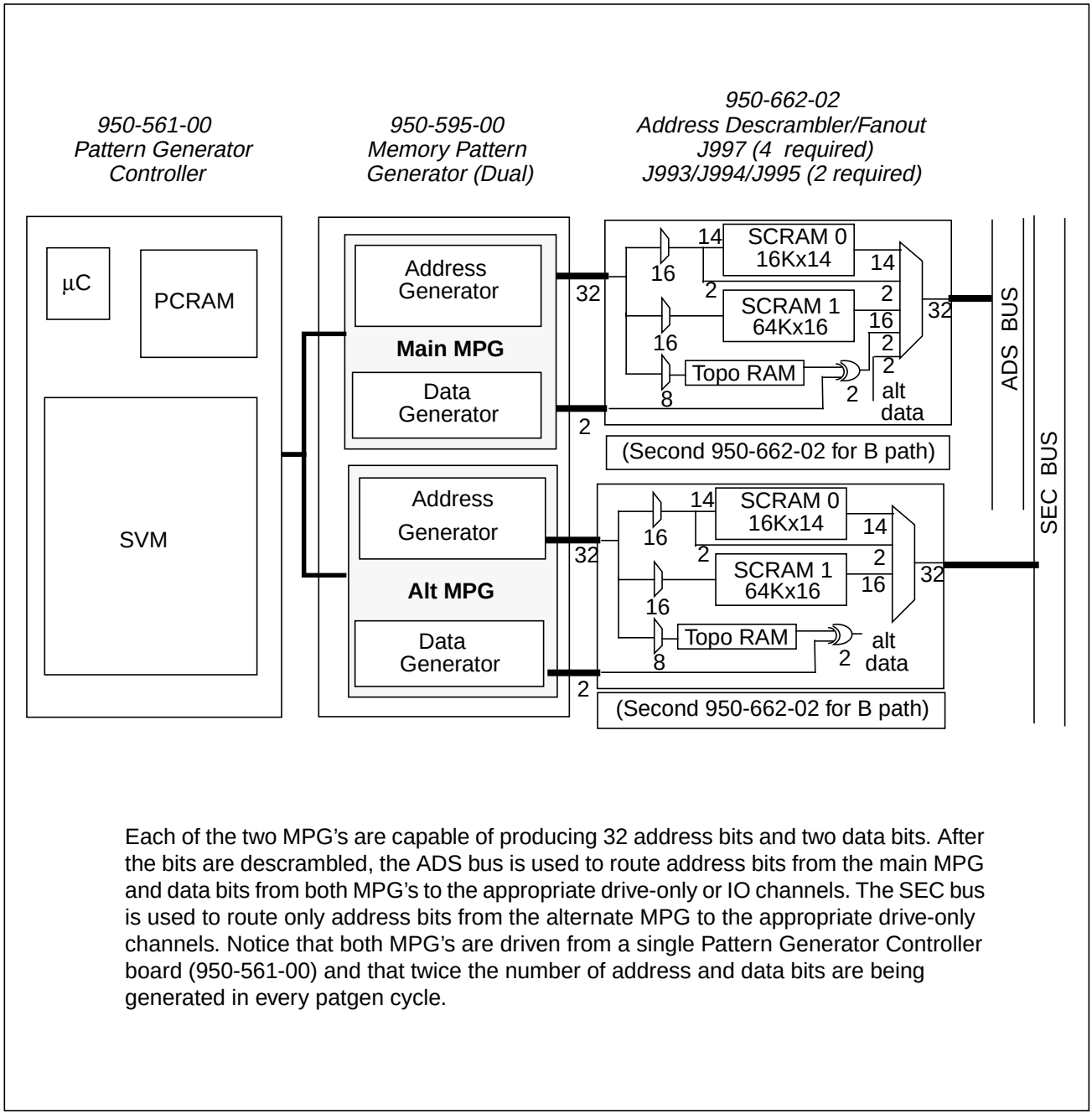
Dual-MPG capabilities can be used to increase device test speeds and to test dual port RAMs.

Many have referred to this operating mode as "dual-patgen", which is misleading. The preferred terminology is "dual-MPG". **Dual-MPG** is a single pattern generator running a single pattern at twice the patgen frequency. **Dual-patgen**, which is officially called dual-asynchronous patgen, is a mode involving two pattern generators running two patterns simultaneously. The standard mode of operation is **single-patgen,** which uses one patgen to run one pattern at the normal patgen frequency.

This chapter describes the requirements and necessary changes for converting an existing single-MPG test program to a dual-MPG test program to effectively double the test frequency. Memory Fail Processing (MFP) software completely supports the dual-MPG operation. Throughout this chapter, there will be references to the MFP templates and function calls. Please refer to chapter 4 of this manual for a complete description of MFP.

What is Dual-MPG?

The concept of dual-MPG is simply to double the frequency by generating not one but two addresses or data bits for each device input or output pin at each patgen cycle. To accomplish this, the dual-MPG must be able to generate enough address and data bits for two device cycles, and secondly, must drive or compare two bits of data using a single channel.



Each of the two MPG's are capable of producing 32 address bits and two data bits. After the bits are descrambled, the ADS bus is used to route address bits from the main MPG and data bits from both MPG's to the appropriate drive-only or IO channels. The SEC bus is used to route only address bits from the alternate MPG to the appropriate drive-only channels. Notice that both MPG's are driven from a single Pattern Generator Controller board (950-561-00) and that twice the number of address and data bits are being generated in every patgen cycle.

Figure 5.1-1: Dual Patgen Overview

The first requirement of dual-MPG is satisfied by acquiring a Dual Memory Pattern Generator (board #950-595-00) and a second Address Descrambler (board #950-662-02). (Note that four Address Descrambler's are required for the J997). The first MPG will be referred to as the "main MPG" and the second as the "alternate MPG". Each MPG generates a single address or data bit for every device input or output pin at every patgen cycle.

To help conceptualize dual-MPG, think of each patgen cycle as being broken into an "early-half" and a "late-half". All the bits generated from the main MPG are the address and data for the "early-half" of the cycle and all the bits generated from the alternate MPG are the address and data for the "late-half" of the cycle. Figure 5.1-2 illustrates this concept.

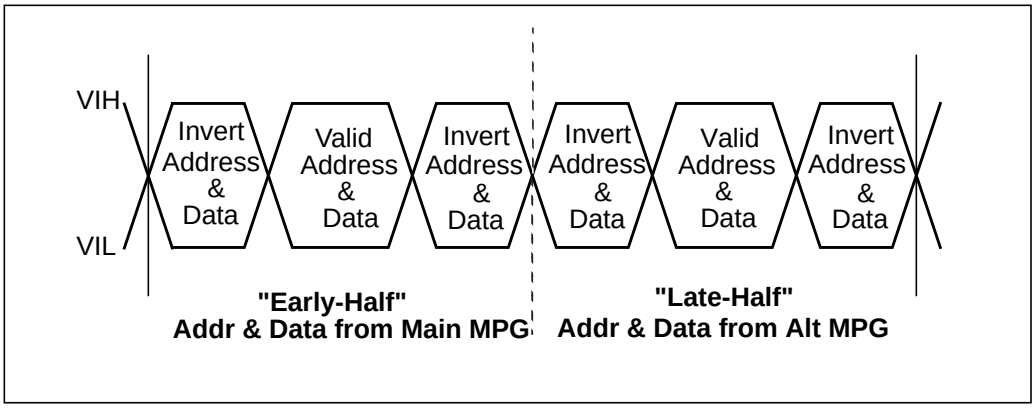


Figure 5.1-2: Dual-MPG with RC, RC[1]

The second requirement of dual-MPG is to drive or compare both bits of address or data in the same cycle on the same channel. This is accomplished by using format muxing on drive-only channels and channel muxing on IO channels. Figure 5.1-3 and Figure 5.1-4 illustrate the concept in reference to dual-MPG mode.

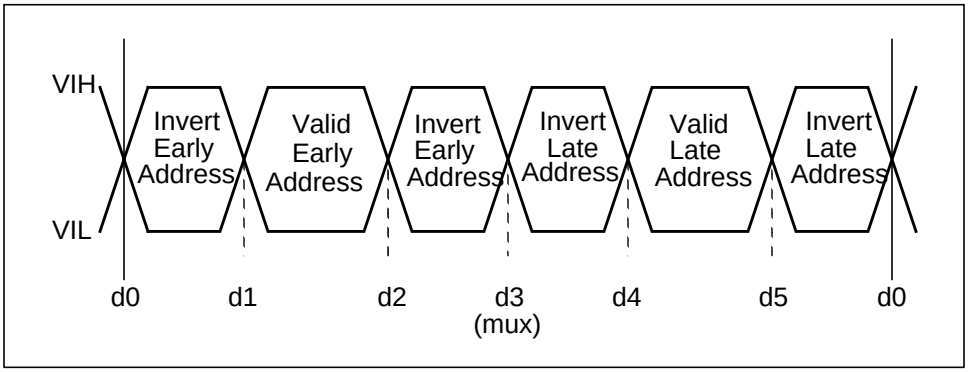


Figure 5.1-3: A/C Format Muxing for Dual-MPG with RC, RC

1. RC = Return Complement

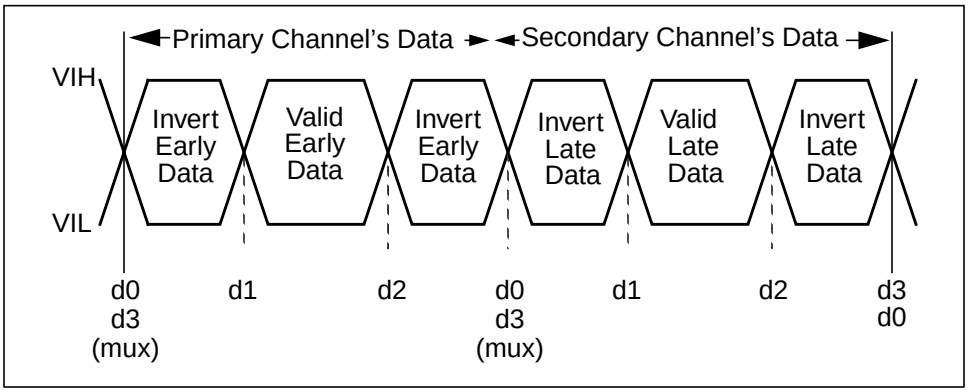


Figure 5.1-4: IO Channel Muxing for Dual MPG with RC, RC

Database Conversion

Given this description of dual-MPG operation, certain test program databases must be converted to allow the test program to run in a dual-MPG mode. The following is a list of the databases which must be altered followed by a description of what needs to be changed in each.

.pinadr	mfp.insadr	.srcadr	.caladr
.chanadr	.waveadr	.svmadr	.wspadr

.pinadr

Every IO pin on your device will now require two IO channels. This is because channel multiplexing will be used to output the two bits of data in one cycle. Each secondary channel's pin name must have a different name than it's primary counterpart but the last digit must be the same for both the primary and the secondary so that the FVM (fail vector memory) will denote failures from either channel as being from the same device pin.

For example: a pin named "DQ0" in a single-MPG environment will now also have a secondary pin named "DQ_0".

For programming purposes, think of this secondary pin in terms of timing control for the "late-half" of the cycle. In fact, all input and output will take place on the primary pin's channel.

.chanadr

For each of the pins in the .pinadr file, a corresponding channel must be assigned to it in the .chanadr file. Due to the hardware restrictions of channel muxing, IO channels must be chosen using the following guideline: **Each set of two muxed IO channels must be adjacent channels where the primary channel is an even numbered IO channel and the secondary channel is the next odd channel.**

The name of the secondary IO channel must be different from the name of the primary IO channel and the last digit must be the same for both the primary and the secondary so that the FVM (fail vector memory) will denote failures from either channel as being from the same device pin.

For example:

```
DQ0     200     ;primary channel for DQ0
DQ_0    201     ;secondary channel for DQ0
```

mfp.insadr

The .mfpadr file consists of a set of MFP template instances. This section discusses the support for dual-MPG that is built into these templates.

MFP Address Scrambling Template

Template Parameter	Description
Which Patgen?	Selects which MPG. Options are Main(0), Alt(1), or Both(2). In dual-MPG mode, "Both" is the appropriate choice since address scrambling will be the same for both MPGs.
Address Pin Names	An address pin will be entered for each address bit that goes to the DUT during a single patgen cycle. Therefore, dual-MPG requires that each address pin be listed twice, once for "early-half" and once for "late-half" of the cycle.
Cycle Position	Acutally refers to whether we are programming the "X-half" (0) or "Y-half" (1) of the cycle for format muxing. For dual-MPG, we will refer to it as "early-half" (0) and "late-half" (1).
Scramble Expression	An expression of patgen bits that will correctly program this address pin during the selected half of the cycle.

Sample MFP Address Scrambling Instance

```
(define-instance mfp_addr_scram   mfp_addr_scram_Template
        (value patgen 2 :choice "Both")
        (value addr_pins
               A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13 A14
               A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13 A14)
        (value suffix
                0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
                1   1   1   1   1   1   1   1   1   1   1   1   1   1   1)
        (value scram_expression
             px0 px1 px2 px3 px4 px5 px6 px7 py0 py1 py2 py3 py4 py5 py6
             px0 px1 px2 px3 px4 px5 px6 px7 py0 py1 py2 py3 py4 py5 py6)
)
```

MFP Bitmap Usage Template

Template Parameter	Description
Number of Sites to Bitmap in Parallel	Specify the number of sites to bitmap n parallel
Store Mask Data	Specify if you want to store mask data (on/off)
Accumulate Bitmaps	Specify if you want to store accumulate data (on/off)

Sample MFP Bitmap Usage Instance

```
(define-instance mfp_bitmap_usage   mfp_bitmap_usage_Template
        (value num_sites 1)
        (value mask 0 :choice "Off")
        (value accum 0 :choice "Off")
)
```

MFP Data Generation Template

Template Parameter	Description
Which Patgen?	Selects which MPG. Options are Main(0), Alt(1), or Both(2). In dual-MPG mode, "Both" is the appropriate choice since data generation and topo inversion will be the same for both MPGs.
Data Pins Names	List only the primary IO channels -- secondary channels are programmed automatically
Cycle Position	Suffix does not apply to dual-MPG for IO pins -- always use cycle "0".
Which dgen?	Select here which dgen (0 or 1) will be routed to the corresponding data pin. All secondary channels (odd numbers channels in the chanmuxed pair) will be assigned the same dgen # but on the alternate MPG.
Invert Expression	Nothing special required for dual-MPG

Sample MFP Data Generation Instance

```
(define-instance mfp_dgen    mfp_dgen_Template
        (value patgen 2 :choice "Both")
        (value data_pins DQS)
        (value suffix 0)
        (value dgen 0)
        (value topo_inv_expr 0)
)
```

MFP MUT Description Template

Template Parameter	Description
Tester Speed in MHz	Dual-MPG requires an entry of 120 or 200 MHz for J997 and 120 MHz for all other memory testers. This informs MFP that you will be using dual-MPG mode.
XY orientation	Normal implies X is rows, Y is columns; flipped is reverse
Rows (# of X addresses)	Total number of rows in the device
Cols (# of Y addresses)	Total number of columns in the device
Number of Output Bits	List only the number of IO pins on the device.
Output Pin Names	List only the primary pin names (the even number channel in the chanmuxed pair).

Sample MFP MUT Description Instance

```
(define-instance mfp_mut_description   mfp_mut_description_Template
        (value speed 200 :choice "200MHz")
        (value orientation 0 :choice "Normal")
        (value rows 256)
        (value cols 128)
        (value num_outputs 8)
        (value output_pins DQ0 DQ1 DQ2 DQ3 DQ4 DQ5 DQ6 DQ7)
)
```

MFP MUT Topology Template

Template Parameter	Description
Which Patgen?	Selects which MPG. Options are Main, Alt, or Both. In dual-MPG mode, "Both" is the appropriate choice since we are describing the MUT topology of a single device but using both MPGs.
Patgen X/Y Bit	List patgen X/Y bits and I/O select bits.
Expression of H/V Bits	Expression of H/V bits determining values for the corresponding X/Y or I/O select bits. H/V bits are the horizontal and vertical address on the bitmap display.

Sample MFP MUT Topology Instance

```
(define-instance mfp_mut_topology   mfp_mut_topology_Template
     (value patgen 2 :choice "Both")
     (value x_y_bit
        px0 px1 px2 px3 px4 px5 px6 px7 py0 py1 py2 py3 py4 py5 py6 iob0)
     (value h_v_expr
        h0  h1  h2  h3   h4  h5  h6  h7  h8  v0  v1  v2  v3  v4  v5  v6)
```

MFP Setup Template

Template Parameter	Description
MUT Description Instance	Name of MUT Description Instance
Capture/Both SCRAM Instance	Name of Address Scrambling Instance
Scan-Only SCRAM Instance	Name of Scan-only address scrambling instance
Data Generator Instance	Name of Dgen Instance
Bitmap Usage Instance	Name of Bitmap Usage Instance
MUT Topology Instance	Name of MUT Topology Instance
Output File	optional: name of database output file
Infile	optional: name of database file to read

Sample MFP Setup Instance

```
(define-instance mfp_setup   mfp_setup_Template
        (value mut_description_instance  mfp_mut_description)
        (value addr_scram_instance            mfp_addr_scram)
        (value scan_scram_instance )
        (value dgen_instance                          mfp_dgen)
        (value bitmap_usage_instance        mfp_bitmap_usage)
        (value mut_topology_instance        mfp_mut_topology)
        (value mfpadr_outfile )
        (value mfpadr_infile  )
)
```

.waveadr

In the beginning of every .waveadr file, the patgen frequency is given. The patgen frequency (referring to the Pattern Generator Controller board) will remain at 60 or 100 MHz for J997 and 60 MHz for all other memory testers.

For example:

(patgen :freq 100MHz)

In the pins-form statement, you must declare all drive-only pins to have format muxing mode turned on. For all IO pins, you must declare channel muxing mode to be turned on. For example:

```
(pins-form
    (:pin     )
    (E        :fmt-mux on)
    (W        :fmt-mux on)
    (G        :fmt-mux on)
    (ADDRS    :fmt-mux on)
    (DQS      :io (zstate d0) :scio off :comp on :strobe edge
              :chan-mux on)
    (DQ_S     :io (zstate d0) :scio off :comp on :strobe edge
              :chan-mux on)
)
```

The timing edges follow the same rules as prescribed by drive-only format muxing and IO channel muxing guidelines. The only difference is the reference to "early-half" and "late-half" of a cycle.

For drive-only channels, the following timing applies:

Edge	Description
d0	Clock in surround data for "early-half" of the cycle
d1	Clock in data for "early-half" of the cycle
d2	Clock in surround data for "early-half" of the cycle
d3	Clock in surround data for "late-half" of the cycle
d4	Clock in data for "late-half" of the cycle
d5	Clock in surround data for "late-half" of the cycle

For IO channels, the following timing applies:

Channel	Edge	Description
Primary	d0	Clock in "early-half" surround data and may be used to change the drive/HIZ state
Primary	d1	Clocks in data for "early-half" of the cycle
Primary	d2	Clock in surround data for "early-half" of the cycle and maybe used to change drive/HiZ state
Primary	d3	Muxes the primary channel to the DUT and may be used to change drive/HIZ state
Primary	c1	Used to start the window strobe for the "early-half" of the cycle
Primary	c2	Used to end window strobe or to place an edge strobe for the "early-half" of the cycle
Secondary	d0	Clock in "late-half" surround data and may be used to change the drive/HIZ state
Secondary	d1	Clocks in data for "late-half" of the cycle
Secondary	d2	Clock in surround data for "late-half" of the cycle and maybe used to change drive/HiZ state
Secondary	d3	Muxes the secondary channel to the DUT
Secondary	c1	Used to start the window strobe for the "late-half" of the cycle
Secondary	c2	Used to end window strobe or to place an edge strobe for the "late-half" of the cycle

For example:

```
        (write_edges          :per "20ns"
                      (ADDRS    rc :fmt1 rc
                                     (d0    "0ns")
                                     (d1    "3ns")
                                     (d2    "8ns")
                                     (d3    "10ns")
                                     (d4    "13ns")
                                     (d5    "18ns")
                       )
                      (DQS      rc    (d0    "0ns")
                                      (d1    "3ns")
                                      (d2    "8ns")
                                      (d3    "0ns")
                                      (c1    "0ns")
                                      (c2    "0ns")
                       )
                      (DQ_S     rc    (d0    "10ns")
                                      (d1    "13ns")
                                      (d2    "18ns")
                                      (d3    "10.5ns")
                                      (c1    "0ns")
                                      (c2    "0ns")
                       )
           )
           (read_edges          :per "20ns"
                      (ADDRS    rc :fmt1 rc
                                     (d0    "0ns")
                                     (d1    "3ns")
                                     (d2    "8ns")
                                     (d3    "10ns")
                                     (d4    "13ns")
                                     (d5    "18ns")
                       )
                      (DQS      nr    (d0    "0ns")
                                      (d1    "0ns")
                                      (d2    "0ns")
                                      (d3    "0ns")
                                      (c1    "8ns")
                                      (c2    "8ns")
                       )
                      (DQ_S    nr    (d0    "0ns")
                                      (d1    "0ns")
                                      (d2    "0ns")
                                      (d3    "0ns")
                                      (c1    "18ns")
                                      (c2    "18ns")
                       )
           )
```

(Note that during a read cycle, the timing edges d0, d1, d2, and d3 have no effect and that only c1 and c2 edges can have any meaning)

.srcadr

This file simply requires the addition of the secondary IO pins in the src-form in order to program a source set for these channels. For dual-MPG mode, the source set of the primary IOs should match the source set of the secondary IOs.

```
(srcset src_set_sram_patterns
        (src-form (:label       E  W  G     ADDRS        DQS         DQ_S)
                  (src_set_1    vm vm vm  (0 0 ads)  (vm vm ads) (vm vm ads))
                  (src_set_init vm vm vm  (0 0 vm)   (vm vm ads) (vm vm ads))
        )
)
```

.svmadr

The most significant change required for dual-MPG mode is in writing the patterns. The patterns that were designed to run in a single-MPG mode must be rewritten for dual-MPG operation.

The essential difference is that each vector is split into two parts -- one to control the main MPG and another to control the alternate MPG. The header names are: "mpg" and "mpg-alt". The information contained in each of the two clauses will independently control each MPG to generate two sets of addresses and data per cycle -- one for "early-half" and one for "late-half".

The mpgset should look like this:

(mpgset sram_patterns **:dual-mpg**

The vector-form should look like this:

```
 (vector-form
     (
             CYCLE_DEFINITION
             :mpg           ;mpg instructions for the main MPG
             :mpg-alt       ;mpg instructions for the alternate MPG
             :op            ;single opcode for the pattern generator
     )
```

The following is an example of a scan pattern written to operate in dual-MPG mode.

```
(default
        INACTIVE_CYCLE :src src_set_1
        (;mpg
        :xreg 0 :yreg 0
        :xa hold :xb hold :xc hold :xd hold
        :ya hold :yb hold :yc hold :yd hold
        :xconst-next hold :yconst-next hold
        :xena-next hold :yena-next hold
        :xaluj zero :xaluk zero :xaluop xor
        :yaluj zero :yaluk zero :yaluop xor
        :xdevadr xa :ydevadr ya :xdevinv off :ydevinv off
        :xdgadr xa :ydgadr ya :xdginv off :ydginv off
        :dgroup-next hold :dset 0
        :catch-clr off
        :disp-push-failcntr off :disp-ign-fails off :disp-clr off
        :mutsim idle :mutsim-inv off
        )
        (;mpg-alt
        :xreg 0 :yreg 0
        :xa hold :xb hold :xc hold :xd hold
        :ya hold :yb hold :yc hold :yd hold
        :xconst-next hold :yconst-next hold
        :xena-next hold :yena-next hold
        :xaluj zero :xaluk zero :xaluop xor
        :yaluj zero :yaluk zero :yaluop xor
        :xdevadr xb :ydevadr yb :xdevinv off :ydevinv off
        :xdgadr xb :ydgadr yb :xdginv off :ydginv off
        :dgroup-next hold :dset 0
        )
        (refresh-disable)
)
```

(continued on next page)

Example (continued)

```
scan
; preset lsb of the x counters to 0 (main) and 1 (alt), y counters to 0
; set the enable masks to mask out the x lsb of both main and alt counters
; preload x counters to xmax, ycounters to ymax
; write 0's xfast through entire array
; read 0's xfast through entire array
; write 1's xfast through entire array
; read 1's xfast through entire array

    (INACTIVE_CYCLE ;preset last bit of counters to 0 (mpg) and 1 (mpg-alt)
                (;mpg
                :xreg 0x0
                :xa jam-reg
                :dgroup-next 0
                )
                (;mpg-alt
                :xreg 0x1
                :xa jam-reg
                :dgroup-next 0
                )
                (nop)
    )

    (INACTIVE_CYCLE ;mask out lsb of counters to inc by 2's
                              ;preset counters max-1(mpg) and max(mpg-alt)
                (;mpg
                :xreg XMAX-1 :yreg YMAX
                :xena-next jam-reg :yena-next jam-reg
                )
                (;mpg-alt
                :xreg XMAX-1 :yreg YMAX
                :xena-next jam-reg :yena-next jam-reg
                )
                (nop)
    )

    (INACTIVE_CYCLE ;mask out lsb of counters to inc by 2's
                              ;preset counters max-1(mpg) and max(mpg-alt)
                (;mpg
                :xreg XMAX-1 :yreg YMAX
                :xa load-reg :ya load-reg
                )
                (;mpg-alt
                :xreg XMAX-1 :yreg YMAX
                :xa load-reg :ya load-reg
                )
                (nop)
     )
```

(continued on next page)

Example (continued)

```
        (WRITE_CYCLE
                (;mpg
                :xa inc :ya inc-link
                :xdgadr xa :ydgadr ya
                :xdevadr xa :ydevadr ya
                :dset 0
                )
                (;mpg-alt
                :xa inc :ya inc-link
                :xdgadr xa :ydgadr ya
                :xdevadr xa :ydevadr ya
                :dset 0
                )
                (rep RPT_FULL_MUX)
        )

        (READ_CYCLE
                (;mpg
                :xa inc :ya inc-link
                :xdgadr xa :ydgadr ya
                :xdevadr xa :ydevadr ya
                :dset 0
                )
                (;mpg-alt
                :xa inc :ya inc-link
                :xdgadr xa :ydgadr ya
                :xdevadr xa :ydevadr ya
                :dset 0
                )
                (rep RPT_FULL_MUX)
        )

        (WRITE_CYCLE
                (;mpg
                :xa inc :ya inc-link
                :xdgadr xa :ydgadr ya
                :xdevadr xa :ydevadr ya
                :dset 1
                )
                (;mpg-alt
                :xa inc :ya inc-link
                :xdgadr xa :ydgadr ya
                :xdevadr xa :ydevadr ya
                :dset 1
                )
                (rep RPT_FULL_MUX)
        )
```

(continued on next page)

Example (continued)

```
        (READ_CYCLE
                (;mpg
                :xa inc :ya inc-link
                :xdgadr xa :ydgadr ya
                :xdevadr xa :ydevadr ya
                :dset 1
                )
                (;mpg-alt
                :xa inc :ya inc-link
                :xdgadr xa :ydgadr ya
                :xdevadr xa :ydevadr ya
                :dset 1
                )
                (rep RPT_FULL_MUX)
        )

        (INACTIVE_CYCLE
                ()()
                (stop)
        )
```

.caladr

The .caladr file must be set up to calibrate drive-only channels with a special format muxing flag and IO channels with a special channel muxing flag (only the primary IO channels should be specified). The patgen frequency (again referring to the Pattern Generator Controller) remains the same.

An example calset

```
(range :level (-2v 7v))

(calset nominal_level_cal
#ifdef J997
        (patgen         :freq 100Mhz)
#else
        (patgen         :freq 60Mhz)
#endif
        (default
                        :vil 0.0
                        :vih 3.0
                        :chan-mux off
                        :fmt-mux off
                        :optimize d0_drive
                        :strobe edge
        )
        (cal-form       (:pin   :vil    :vih    :fmt-mux)
                        (CLKS   0.0     3.0     on)
                        (ADDRS  0.0     3.0     on)
        )
        (cal-form       (:pin   :strobe :chan-mux )
                        (DQS    edge    on        )
        )
)
```

.wspadr

The workspace should require no changes if you are already using MFP in your test program. However, if MFP has not yet been incorporated into your test program you must make the necessary additions. (See chapter 4 of this manual for a full description of how to incorporate MFP into your test plan.)

IG900+ Tool Support

Several IG900+ tools have been enhanced to support dual-MPG mode. The following is a description of what these enhancements are and how to use them.

Patedit

The only change to patedit is the addition of the field :mpg-alt and consequently all the MPG instructions for the alternate MPG by SVM address. The information should reflect the information from your .svmadr file. As with the main MPG instructions, the alternate MPG instructions can be modified via the patedit tool.

Patdebug

Patdebug has three new fields to display the X, Y, and data generator information for the alternate MPG. These fields are named "Mpg Alt X Addr", "Mpg Alt Y Addr", and "Mpg Alt Data" respectively. Display of this information can be controlled from the "View"/"Layout" button from patdebug's main panel.

Figure 5.1-5 shows a partial output from a scan pattern. Notice that 3 additional fields, "Alt_X.D", "Alt_Y.D", and "AltDG.D".

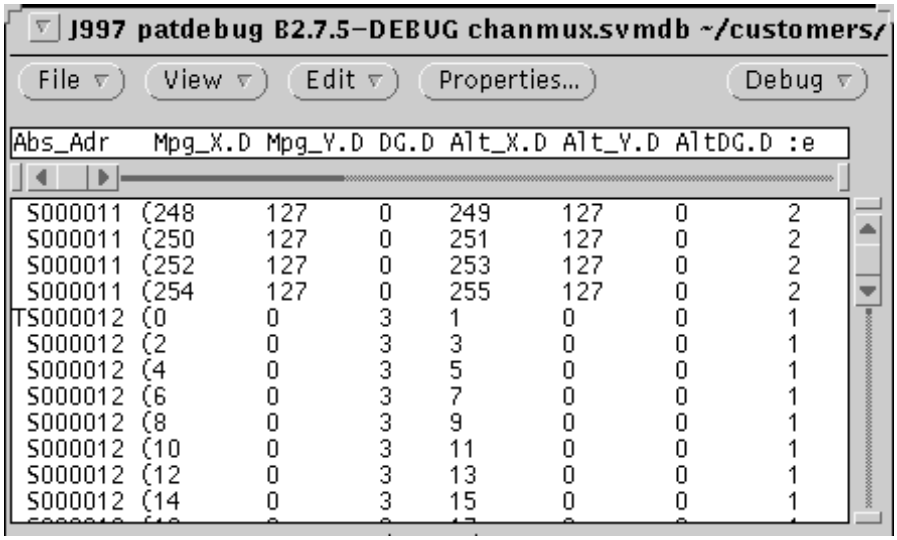


Figure 5.1-5: Patdebug Menu

Wavetool

Wavetool's only changes are a result of using format muxing and channel muxing. Consequently, each drive-only channel shows two pieces of data driven in a single cycle. Each primary IO channel also shows two pieces of data driven (or compared) on the same cycle. The secondary IO channel shows only the "late-half" data. This follows the concept of dual-MPG since all the input/output occurs on the primary IO channels and the secondary is muxed in only to drive/compare the "late-half" data.

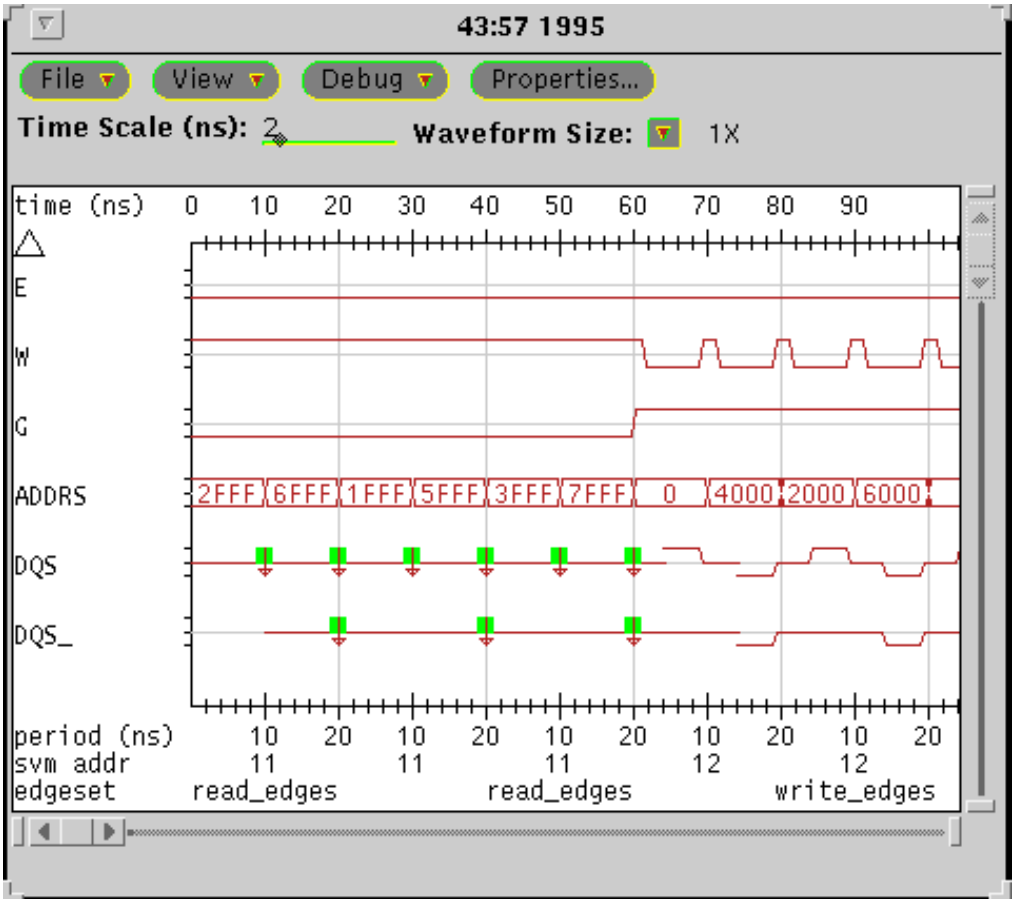


Figure 5.1-6: Wavetool Display

Hazards

Since dual-MPG is a patgen mode, it violates no system timing hazards. Therefore, programmers should pay close attention to hazard warnings. The most common timing violation when using dual-MPG mode to achieve increased test frequency is the minimum pulse width spec. Keep in mind that both data pulses and surround data pulses must meet this spec.

Offline

When loading a dual-MPG test program offline, you will normally receive a warning message telling you that the system (offline model) does not have an SEC bus. While this will have no impact on how the test program executes, it can be easily fixed by adding a ***-w*** flag to the flag block of the workspace file.

For example:

```
(flag
        (flag-form
                (:name :elements :type)
                (tflags "-w -m" tldflags) ; need -w for off-line sim.
;               (tflags "-m"    tldflags)
                (lflags "-g"    ldflags)
        )
)
```

Index

Symbols

(define_template ...) statement *3.2.5*
.*adr *1.1.3*
.*db *1.1.3*
.caladr *1.1.6*, *1.9.1*
.caladr files
  dual-mpg, programming for *5.1.18*
.caldb *1.1.6*, *1.9.1*
.chanadr *1.1.5*, *1.3.1*
.chanadr files
  dual-mpg, programming for *5.1.5*
.chandb *1.1.5*
.dgenadr *1.1.6*, *1.8.3*
.dgendb *1.1.6*
.insadr *1.1.5*, *1.11.2*
.leveladr *1.1.5*, *1.6.2*
.leveldb *1.1.5*
.mfpadr files
  dual-mpg, programming MFP templates *5.1.5*
.mutadr *1.1.5*
.mutdb *1.1.5*
.pinadr *1.1.5*, *1.2.2*
.pinadr files
  dual-mpg, programming for *5.1.4*
.pindb *1.1.5*, *1.2.2*
.specadr *1.1.5*, *1.5.2*
.specdb *1.1.5*
.srcadr *1.1.6*, *1.8.6*
.srcadr files
  dual-mpg, programming for *5.1.13*
.srcdb *1.1.6*
.startadr *1.1.6*, *1.8.22*
.startdb *1.1.6*
.svmadr *1.1.6*, *1.8.8*
.svmadr files
  dual-mpg, programming for *5.1.13*
.svmdb *1.1.6*
.t file (customizing templates) *3.2.9*
.tdl file (customizing templates) *3.2.3*
.waveadr *1.1.6*
.waveadr files
  dual-mpg, programming for *5.1.10*
.wavedb *1.1.6*
.wspadr *1.1.6*, *1.10.3*
.wspadr files
  dual-mpg, programming for *5.1.18*
.wspdb *1.1.6*, *1.10.3*

Numerics

100 MHz mode *2.1.3*
200 MHz Mode *2.1.5*
60 MHz mode *2.1.4*

A

accum parameter *4.2.8*
addr_pins parameter *4.2.2*
addr_scram_instance parameter *4.2.22*
address counter *2.1.11*
address generation *2.1.11*
address scrambling *1.4.1*, *2.1.11*, *2.1.14*
adjust-until parameters *1.5.1*
ADRedit tool *1.1.4*
ADS bus *2.1.7*
AEL *1.9.1*
arithmetic logic units (ALUs) *2.1.13*
AT900 *1.1.3*, *1.12.3*
autotitle (instance parameter) *3.1.5*

B

background_Template *1.11.3*, *3.1.4*
bias supply *2.1.53*
bitmap_usage_instance parameter *4.2.23*
build_workspace *1.10.6*

C

caladr *1.9.2*
Calcsheet *1.1.4*, *1.6.1*
caldb *1.9.2*
calibration *1.9.1*
calset (instance parameter) *3.1.21*
calsets *1.1.6*, *1.9.1*
channel database *1.3.1*, *1.3.5*

channel multiplexing 2.1.5, 2.1.25, 3.1.16
chansets 1.1.5
columns parameter 4.2.13
comparators 3.1.17, 3.1.18, 3.1.26
  with ICC tests 3.1.34
continuity tests 3.1.6
corner_list_Template 1.11.3
cumulative reject (CR) latch 3.1.18
current monitor
  with ICC tests 3.1.31, 3.1.31–3.1.32, 3.1.34

D

data generator 2.1.11, 2.1.15
data generator database example 1.8.5
data sheet 3.1.15, 3.1.24
data_pins parameter 4.2.10
datalog output 3.1.31
datasheet_Template 1.11.4
device output protection diodes 3.1.24
device pins 1.2.1, 1.3.4
device shape 1.4.1
device size 1.4.1
device topology 1.4.1
dgen parameter 4.2.11
dgen_instance parameter 4.2.23
dgensets 1.1.6
DIB (device interface board)
  DSR (device sense return) connections 3.1.31
  wiring for ICC tests 3.1.31
dotests (instance parameter) 3.1.21
dotime (instance parameter) 3.1.5
drive-only 2.1.42
drive-only channels 1.3.3, 2.1.49
driver channel numbers according to site/function
  J993, J995, J997 1.3.7
driver0 (instance parameter) 3.1.10
driver1 (instance parameter) 3.1.10
driveroff (instance parameter) 3.1.10
drivers 3.1.15, 3.1.26
  formats 2.1.19
  with ICC tests 3.1.33
dual-mpg
  .caladr programming for 5.1.18
  .chanadr programming for 5.1.5
  database conversion 5.1.4
  definition of 5.1.1
  explanation of 5.1.2
  mfp.insadr programming for 5.1.5
  offline usage 5.1.21
  .pinadr programming for 5.1.4
  programming with 5.1.1
  .srcadr programming for 5.1.13
  .svmadr programming for 5.1.13
  timing edges 5.1.10
  .waveadr programming for 5.1.10
  .wspadr programming for 5.1.18
dual-patgen
  definition of 5.1.1
duringpat (instance parameter) 3.1.36
dynamic current load 2.1.51
dynamic load circuits 3.1.17, 3.1.26
  programming 3.1.19

E

edge strobe 3.1.17, 3.1.18
  defined 3.1.18
edge transitions See timing edges, t1 timing edge, t2 timing edge, t3 timing edge
eqset statement 1.1.5, 1.1.6, 1.6.1, 1.6.3
equation-form 1.6.3
error detector 3.1.13, 3.1.17
example flow file 1.12.4
EXECUTE macro 1.12.2
Executive 1.1.3

F

fail processor 3.1.13, 3.1.17
FAIL_DO macro 1.12.3
FET matrix 3.1.25
fload (instance parameter) 3.1.9
flow control 1.1.1, 1.1.3, 1.12.1
for function 3.2.16
forcespec1 (instance parameter) 3.1.9
forcespec2 (instance parameter) 3.1.9
forceval1 (instance parameter) 3.1.9
format-Muxing 2.1.24
formatting system 3.1.15, 3.1.16, 3.1.17
func_Template 1.11.3
functional system 3.1.13, 3.1.26
  with leakage tests 3.1.27
functional tests 3.1.13

G

guidelines for timing 2.2.1

H

h_v_expr parameter 4.2.15
Hazards 5.1.20
high-impedance output current leakage 3.1.24
  See also leakage tests
hilimspec (instance parameter) 3.1.30
hilimval (instance parameter) 3.1.9

I

I/O switching
  with t3 timing edge 3.1.16
ICC tests 3.1.31
  static 3.1.31, 3.1.33
  with current monitor 3.1.31–3.1.32
  with system PMU 3.1.31, 3.1.33
impedance lock 2.1.51
import statement 1.12.2
input current leakage 3.1.24
  See also leakage tests
input system 3.1.13
instance databases 1.11.2
  basic steps to creating 3.1.2
instances 1.11.1
  creating, basic steps 3.1.2
Insted 1.1.4, 1.11.2, 1.11.5
IO channels 1.3.3, 2.1.50
IO compare modes 2.1.28
IO Switching Modes 2.1.30
IOH 2.1.51
IOL 2.1.51

irange (instance parameter) 3.1.9
ivs_pmu_Template 1.11.4
ivs_Template 1.11.4
ivspin (instance parameter) 3.1.37

J

job_main function 1.12.1, 1.12.2
job-driven bitmapping 4.3.4

L

leakage tests 3.1.24
  defined 3.1.24
  with the pin PMU 3.1.26
  with the system PMU 3.1.25
level database example 1.6.4
levels (instance parameter) 3.1.10
logic function generators (LFGS) 2.1.15
lolimspec (instance parameter) 3.1.30
lolimval (instance parameter) 3.1.9

M

mach2 mode 2.1.27
macro definitions 4.4.11
macros 3.2.10
  EXECUTE 1.12.2
  FAIL_DO 1.12.3
  PASS_DO 1.12.3
main.o 1.1.5
main.t 1.1.5
makefile 1.1.1, 1.1.3, 1.1.6
mask parameter 4.2.8
measure (instance parameter) 3.1.30
memory fail processing
  accumulate mode 4.1.3
  ADR files 4.1.4
  ADR translators 4.4.12
  high-level drivers 4.1.3
  high-level functions 4.3.2
  high-level programming 4.1.3
  hldsets 4.1.5
  low-level programming 4.1.4, 4.4.1
  mask mode 4.1.3
  parallel capture 4.1.3
  Setup Template 4.1.2
  using the software 4.1.1
  workspace requirements 4.1.5
Memory Fail Processing, See MFP
MFP
  address scrambling template 5.1.5
  bitmap usage template 5.1.6
  data generation template 5.1.7
  MUT description template 5.1.7
  MUT topology template 5.1.8
  setup template 5.1.9
MFP Address Scramble Template 4.2.2–4.2.4
  example instance 4.2.4, 4.2.7, 4.2.21
MFP Address Scramble Template 2 4.2.6, 4.2.7
MFP Bitmap Usage Template 4.2.8–4.2.9
  example instance 4.2.9
MFP Dgen Template
  example instance 4.2.12
MFP MUT Description Template 4.2.13–4.2.14
  example instance 4.2.14
MFP MUT Topology Template 4.2.15–4.2.19
  example instances 4.2.16–4.2.19
MFP MUT Topology Template 2 4.2.20
  example 4.2.21
MFP see also memory fail processing
MFP Setup Template 4.2.22–4.2.25
  example instance 4.2.25
mfp.insadr files
  dual-mpg, programming for 5.1.5
mfpadr_infile parameter 4.2.24
mfpadr_outfile parameter 4.2.24
minimum pulse width, guidelines 2.2.1
mpause (instance paramter) 3.1.10
MPG (Memory Pattern Generator)
  dual-mpg
    programming of 5.1.1
    See also dual-mpg
mpgset statements 1.8.8
MUT database file 1.4.1
mut_description_instance parameter 4.2.22
mut_topology_instances parameter 4.2.24
mutsets 1.1.5, 1.4.5

N

non-return, See NR
NR
  guidelines for 2.2.12
num_outputs parameter 4.2.13
num_sites parameter 4.2.8

O

offline, loading dual-mpg test programs offline 5.1.21
orientation parameter 4.2.13
output current leakage 3.1.24
  See also leakage tests
output system 3.1.13
output_pins parameter 4.2.13
overlays 1.13.1

P

parallel capture, memory fail processing 4.1.3
parameters
  accum parameter 4.2.8
  addr_pins parameter 4.2.2
  addr_scram_instance parameter 4.2.22
  bitmap_usage_instance parameter 4.2.23
  columns parameter 4.2.13
  data_pins parameter 4.2.10
  dgen parameter 4.2.11
  dgen_instance parameter 4.2.23
  h_v_expr parameter 4.2.15
  mask parameter 4.2.8
  mfpadr_infile parameter 4.2.24
  mfpadr_outfile parameter 4.2.24
  mut_description_instance parameter 4.2.22
  mut_topology_instances parameter 4.2.24
  num_outputs parameter 4.2.13
  num_sites parameter 4.2.8
  orientation parameter 4.2.13
  output_pins parameter 4.2.13
  patgen parameter 4.2.2, 4.2.6, 4.2.10, 4.2.15, 4.2.20
  rows parameter 4.2.13
  scram_expression parameter 4.2.3
  speed parameter 4.2.13

suffix parameter 4.2.2, 4.2.11
topo_inv_expr parameter 4.2.11
x_y_bit parameter 4.2.15
PASS_DO macro 1.12.3
Patdebug 5.1.19
Patedit 1.1.4, 5.1.19
patgen parameter 4.2.2, 4.2.6, 4.2.10, 4.2.15, 4.2.20
pattern (instance parameter) 3.1.21
pattern flow 2.1.10
pattern generator 2.1.6, 3.1.15, 3.1.17
with dynamic ICC tests 3.1.32, 3.1.34
pattern scaling 1.8.19
pattern sync conditions 2.1.56
period 3.1.15, 3.1.16
pin database file 1.2.1
pin groups 1.2.1
pin PMU 2.1.54, 3.1.25, 3.1.26
current range settings 3.1.26
described 3.1.26
programming 3.1.26
voltage range settings 3.1.26
with leakage tests 3.1.24
with system PMU shorts and opens tests 3.1.7
pin_pmu_alternate_Template 1.11.4
pin_pmu_Template 1.11.4
pin-index keys (PIKs) 2.1.52
pinlist (instance parameter) 3.1.9, 3.1.30
pins statement 1.6.1
pinset statement 1.2.2
pinsets 1.1.5
PMU, see pin PMU or system PMU
postbody (instance parameter) 3.1.11
prebody (instance parameter) 3.1.11
program flow 1.11.1
create flow file 1.1.3
programmable current load 2.1.51

R

rows parameter 4.2.13
runtime expressions 1.8.19

S

same cycle IO 2.1.37
scaling functions 1.8.19
SCIO 2.1.37
SCIO mode 2.1.37
scram_expression parameter 4.2.3
SEC bus 2.1.7
sequence (instance parameter) 3.1.10, 3.1.29
Shmoo 1.5.1
shmoo axis ranges 1.5.1
shorts and opens tests 3.1.6, 3.1.19
with the system PMU 3.1.7
signal names 1.3.4
single-patgen
definition of 5.1.1
source select 2.1.35
source select bits 2.1.33
source select database examples 1.8.8
source select memory 2.1.32
source select RAM 2.1.33
spec limits 1.5.1
specification database example 1.5.4
specification database files 1.5.1
specset statement 1.1.5, 1.5.1, 1.5.2, 1.5.3
specsets 1.1.5
Specsheet 1.1.4, 1.5.1
speed parameter 4.2.13
srcset statement 1.1.6, 1.8.6
start patgen statement 3.1.15
startset statement 1.8.22
startsets 1.1.6
stop_on_first_fail (instance parameter) 3.1.21
Subroutine Vector Memory (SVM) 2.1.8
suffix parameter 4.2.2, 4.2.11
SVM database example 1.8.13
svmsets 1.1.6
system block diagram 2.1.2
system PMU 2.1.55
current range settings 3.1.25
described 3.1.25
voltage range settings 3.1.25
with ICC tests 3.1.31, 3.1.33–3.1.34
with leakage tests 3.1.24, 3.1.25
with shorts and opens tests 3.1.7

T

t0 timing edge See period
t1 timing edge 3.1.16
t2 timing edge 3.1.16
t3 timing edge 3.1.16
TEMPLATE_ABORT (template macro) 3.2.23
TEMPLATE_BODY_BEGIN (template macro) 3.2.22
TEMPLATE_BODY_END (template macro) 3.2.22
TEMPLATE_ID (template macro) 3.2.23
TEMPLATE_POST_BODY_BEGIN (template macro) 3.2.22
TEMPLATE_POST_BODY_END (template macro) 3.2.22
TEMPLATE_PRE_BODY_BEGIN (template macro) 3.2.22
TEMPLATE_PRE_BODY_END (template macro) 3.2.22
TEMPLATE_RETURN (template macro) 3.2.22
templates 1.11.1
background_Template 1.11.3, 3.1.4
datasheet_Template 1.11.4
func_Template 1.11.3
ivs_pmu_Template 1.11.4
ivs_Template 1.11.4
MFP Address Scramble 4.2.2–4.2.4
MFP Address Scramble 2 4.2.6
MFP Bitmap Usage 4.2.8–4.2.9
MFP Dgen 4.2.12
MFP Dgen Template 4.2.10
MFP MUT Description 4.2.13–4.2.14
MFP MUT Topology 4.2.15–4.2.19
MFP MUT Topology 2 4.2.20
MFP Setup 4.2.22–4.2.25
pin_pmu_alternative_Template 1.11.4
pin_pmu_Template 1.11.4
test system CPU 3.1.15
threshold voltage (VT) 3.1.19
timing edges 2.1.18, 3.1.15
drive-only, purpose of each edge 2.2.10
dual-mpg, drive-only channels 5.1.10

dual-mpg, IO channels 5.1.11
IO, purpose of each edges 2.2.12
See also period, t1 timing edge, t2 timing edge, t3 timing edge
See also timing guidelines
timing guidelines 2.2.1
timing system 3.1.15, 3.1.16, 3.1.17
Tlib_alarmcheck (template function) 3.2.30
TLIB_AUTOTITLE (template macro) 3.2.24
TLIB_CALDATA (template macro) 3.2.23
TLIB_CHECK_DISABLE (template macro) 3.2.25
Tlib_clearstop (template function) 3.2.31
TLIB_DEB (template macro) 3.2.26
Tlib_debug (template variable) 3.2.27
TLIB_EPR (template macro) 3.2.26
Tlib_floatpins (template function) 3.2.29
TLIB_FORCETEMPLATEFAIL (template macro) 3.2.23
Tlib_get_time (template function) 3.2.31
TLIB_HASVALUE (template macro) 3.2.25
TLIB_LEVEL (template macro) 3.2.23
Tlib_maxforce (template variable) 3.2.27
Tlib_minforce (template variables) 3.2.27
TLIB_NEXT (template macro) 3.2.26
TLIB_PATSTART (template macro) 3.2.25
TLIB_PATSTOP (template macro) 3.2.25
Tlib_pin_pmu_check_irange (template function) 3.2.31
Tlib_pmu_check_irange (template function) 3.2.30
Tlib_pmu_check_vrange (template function) 3.2.30
Tlib_pmu_rngstring (template function) 3.2.28
TLIB_POSTTEST (template macro) 3.2.24
TLIB_PRETEST (template macro) 3.2.24
Tlib_restore_all_vs (template function) 3.2.30
Tlib_restore_levels (template function) 3.2.30
Tlib_save_all_vs (template function) 3.2.30
Tlib_save_levels (template function) 3.2.30
Tlib_set_pmu_force_bounds 3.2.28
Tlib_set_pmu_force_bounds (template function) 3.2.28
Tlib_set_ppmu_force_bounds (template function) 3.2.29
Tlib_setstop (template function) 3.2.31
Tlib_specs_modified 3.2.27
Tlib_specs_modified (template variable) 3.2.27
Tlib_start_timer (template function) 3.2.31
Tlib_stop_timer (template function) 3.2.31
TLIB_TITLE (template macro) 3.2.24
Tlib_tnumarray (template array) 3.2.27
Tlib_trap (template function) 3.2.30
Tlib_unfloatpins (template function) 3.2.29
TLIB_WAVEFORM (template macro) 3.2.24
Tlib_zero_all_vs (template function) 3.2.30
Tlib_zero_levels (template function) 3.2.30
tnum2 (instance parameter) 3.1.9
topo_inv_expr parameter 4.2.11
topological data inversion 1.4.1
tri-state output current leakage 3.1.24
See also leakage tests
tt_mfp_adr_map_pseudo_sites 4.4.2
tt_mfp_capture_completed 4.3.9
tt_mfp_capture_setup 4.3.8
tt_mfp_clear_display_ram 4.4.9
tt_mfp_clear_fmm 4.4.9
tt_mfp_disable_fmm 4.4.10
tt_mfp_do_clear 4.3.7
tt_mfp_do_scan 4.3.11
tt_mfp_file_checks_off 4.3.16
tt_mfp_file_checks_on 4.3.16
tt_mfp_get_es 4.3.16
tt_mfp_get_fail_cnt 4.4.9
tt_mfp_get_saved_fc 4.4.9
tt_mfp_get_waveform 4.3.16
TT_MFP_HLD_TYPE 4.4.3
TT_MFP_HLD_TYPE_ALL 4.4.4
TT_MFP_HLD_TYPE_CATCHRAM 4.4.3
TT_MFP_HLD_TYPE_DISPLAY_PROC 4.4.3
TT_MFP_HLD_TYPE_FAILMAP 4.4.3
TT_MFP_HLD_TYPE_MODULE 4.4.3
TT_MFP_HLD_TYPE_RA 4.4.4
TT_MFP_HLD_TYPE_RA_FORM 4.4.4
TT_MFP_HLD_TYPE_RA_MUX 4.4.3
TT_MFP_HLD_TYPE_RFL_BUS 4.4.3
TT_MFP_HLD_TYPE_RFL_MUX 4.4.3
TT_MFP_HLD_TYPE_SCRAM 4.4.3
TT_MFP_HLD_TYPE_TOPO 4.4.3
tt_mfp_lowlevel_adr_capture_setup 4.4.2, 4.4.3
tt_mfp_lowlevel_adr_scan_setup 4.4.2, 4.4.3
tt_mfp_lowlevel_capture_setup 4.4.2, 4.4.3
tt_mfp_lowlevel_scan_setup 4.4.2, 4.4.3
tt_mfp_map_pseudo_site 4.3.6
TT_MFP_OP_TYPE, MFP arguments 4.3.5
TT_MFP_OUTMUX_BIT 4.4.7
tt_mfp_rect 4.4.11
tt_mfp_save_fail_cnt 4.4.9
tt_mfp_scan_completed 4.3.11
tt_mfp_scan_fmm_dr 4.4.10
tt_mfp_scan_setup 4.3.9
tt_mfp_set_data_mux 4.4.10
tt_mfp_set_es 4.3.16
tt_mfp_set_inmux 4.4.5
tt_mfp_set_outmux 4.4.7
tt_mfp_set_topo_block 4.4.11
tt_mfp_set_waveform 4.3.16
tt_mfp_test_func_setup 4.3.7
TT_MFP_TOPO_INV, MFP arguments 4.3.5

U

use pinset statement 1.6.2

V

Vch 2.1.51
Vcl 2.1.51
vector opcodes 3.1.15
vectors 3.1.15
voltage sources (VS) 2.1.53, 3.1.13, 3.1.26, 3.1.32, 3.1.33
buffer 3.1.31
VS buffer See voltage sources (VS)
VS See voltage sources (VS)
vs-form 1.6.3

W

waveform database example 1.7.4
waveform database file 1.7.2
waveform formats 3.1.15, 3.1.17–3.1.18
See also timing edges, vectors, waveset formats, waveset formatting
waveform generators 2.1.17
waveforms 3.1.15, 3.1.17, 3.1.17–3.1.18
waves (instance parameter) 3.1.22
waveset address 3.1.15

waveset formats *3.1.15*, *3.1.16*
waveset formatting *3.1.16*
waveset timing *3.1.16*
Wavetool *5.1.20*
window strobe *3.1.17–3.1.18*
  defined *3.1.18*
workspace *1.1.1*, *1.1.3*
workspace database *1.10.1*
  example *1.10.7*
writesys(patgen_start) *3.1.15*
writesys(patgen_stop) *3.1.15*
writesys(patgen_stop_mode) *3.1.15*

X

X address values *2.1.11*
x_y_bit parameter *4.2.15*

Y

Y address values *2.1.11*